1992

Parallel algorithms for Hough transform

Fevzi Oktay Ozbek
Lehigh University

AUTHOR:

Ozbek, Fevzi Oktay

TITLE:

Parallel Algorithms for Hough Transform

DATE: May 31, 1992

PARALLEL ALGORITHMS FOR HOUGH TRANSFORM

by

FEVZI OKTAY OZBEK

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science in Electrical Engineering

Lehigh University
Bethlehem, Pennsylvania
1992

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

May 14, 1992

Date

Advisor in Charge

EECS Department Chairperson

Acknowledgements

I would like to acknowledge the guidance and support of my advisor, Professor Meghanad D. Wagh. This thesis was possible because of his patience and great knowledge.

I would also like thank to all my friends in Lehigh University.

Table of Contents

Abstract	**1**
1. Introduction	**2**
1.1. Overview	2
1.2. Organization of the Thesis	5
2. Hough Transform and Parallel Processing	**6**
2.1. Hough Transform	6
2.2. Various Applications of the Hough Transform	10
2.3. Parallel Processing	12
2.4. Hypercube Parallel Computers	14
2.5. NCUBE/10 Parallel Computer	18
2.6. Communication in NCUBE/10	20
2.7. Binary Images Used	20
3. Implementations on a Linear Array	**22**
3.1. Introduction	22
3.2. Embedding a Linear Array in Hypercube	23
3.3. Implementation	25
3.4. Results	30
4. Implementations on a Mesh	**38**
4.1. Introduction	38
4.2. Embedding a Mesh in Hypercube	40
4.3. Implementation	41
4.4. Results	46
5. Implementations on a Hypercube	**53**
5.1. Introduction	53
5.2. Implementation	54
5.3. Results	58
6. Conclusion	**64**
6.1. Discussion	64
6.2. Future Directions	65
References	**66**
Vita	**69**

List of Figures

Figure 2.1.(a).	3 collinear points in the image space.	8
Figure 2.1.(b).	Parameter space generated by 3 colinear points.	9
Figure 2.2.(a).	Hypercubes of degree 0,1 and 2.	15
Figure 2.2.(b).	Hypercube of degree 3.	15
Figure 2.2.(c).	Hypercube of degree 4.	16
Figure 3.1.	Linear Array with n processors.	22
Figure 3.2.	Allocation of processors used in Linear Array.	28
Figure 3.3.(a).	Implementation of 4 processor Linear Array.	28
Figure 3.3.(b).	Implementation of 7 processor Linear Array.	28
Figure 3.3.(c).	Implementation of 11 processor Linear Array.	29
Figure 3.3.(d).	Implementation of 16 processor Linear Array.	29
Figure 3.3.(e).	Implementation of 31 processor Linear Array.	29
Figure 3.4.(a).	Dependence of Time on image distribution Standard Linear Array / 31 processors.	30
Figure 3.4.(b).	Dependence of Time on image distribution Modified Linear Array / 31 processors.	31
Figure 3.5.(a).	Dependence of Speed-up on number of processors. Image densities: 5%, 10%, 15%.	33
Figure 3.5.(b).	Dependence of Speed-up on number of processors. Image densities: 20%, 25%.	34
Figure 3.6.	Dependence of efficiency on image density Modified Linear Array Algorithm.	36
Figure 4.1.	Mesh with NxN processors.	39
Figure 4.2.(a).	Implementation of the Algorithm with 4 processors.	42
Figure 4.2.(b).	Implementation of the Algorithm with 7 processors.	43
Figure 4.2.(c).	Implementation of the Algorithm with 11 processors.	43
Figure 4.2.(d).	Implementation of the Algorithm with 16 processors.	44
Figure 4.2.(e).	Implementation of the Algorithm with 31 processors.	45
Figure 4.3.(a).	Dependence of Time on image distribution Standard Mesh / 31 processors.	46

Figure 4.3.(b).	Dependence of Time on image distribution Modified Mesh / 31 processors.	47
Figure 4.4.(a).	Dependence of Speed-up on number of processors for image densities: 5%,10%,15%.	48
Figure 4.4.(b).	Dependence of Speed-up on number of processors for image densities: 20%,25%.	49
Figure 4.5.	Dependence of efficiency on image density Mesh / Modified Algorithm.	52
Figure 5.1.(a).	Implementation of Algorithm with 4 processors.	54
Figure 5.1.(b).	Implementation of Algorithm with 7 processors.	55
Figure 5.1.(c).	Implementation of Algorithm with 11 processors.	55
Figure 5.1.(d).	Implementation of Algorithm with 16 processors.	56
Figure 5.1.(e).	Implementation of Algorithm with 31 processors.	57
Figure 5.2.(a).	Dependence of Time on image distribution Hypercube / Standard Algorithm / 31 processors.	58
Figure 5.2.(b).	Dependence of Time on image distribution Hypercube / Modified Algorithm / 31 processors.	59
Figure 5.3.(a).	Dependence of Speed-up on number of processors. Image densities: 5%,10%,15%.	60
Figure 5.3.(b).	Dependence of Speed-up on number of processors. Image densities: 20%, 25%.	61
Figure 5.4.	Dependence of efficiency on image density Hypercube / Modified Algorithm.	63

List of Tables

Table 3.1.	8 node Linear Array mapping for a hypercube of degree 3 using Gray Code.	25
Table 3.2.	Percent Improvement in Speed-up due to modified Algorithm.	35
Table 4.1.	Embedding 16 node mesh array in hypercube of degree 4.	41
Table 4.2.	Percent Improvement in Speed-up due to modified Algorithm.	51
Table 5.1.	Percent Improvement in Speed-up due to modified Algorithm	62

Abstract

This Thesis describes two parallel algorithms for Hough Transform and analyzes the performance of these algorithms on three different architectures, linear array, mesh and hypercube. The experiments were performed on the NCUBE/10 hypercube at Lehigh University. The results show that the performance of the algorithms are not liable to asynchronism overheads, which result from nonhomogeneous distribution in the images. The dynamic image partitioning scheme developed in this work is shown to outperform the static partitioning in all architectures and for all images.

Chapter 1

Introduction

1.1. Overview

Image processing and pattern recognition techniques are finding more and more applications in industrial automation, health care and military. Hough Transform is one of the most popular forms of Pattern Recognition Technique. When used to detect straight lines, it transforms 2-dimensional image space into 2-dimensional parameter space. Depending on the parameters chosen to represent a straight line, the basic equation used for Hough Transform varies. Hough Transform is extremely computation intensive because it generates a curve in the parameter space for each point in the image space. For an average image size, it is usually required to evaluate the basic equation and update the parameter space more than a million times. This puts a great stress on the computational mechanism. It should be also noted that some applications like motion guidance systems in robots require a stream of image frames to be processed instead of one frame, thus increasing the need for computational power even more.

In recent years, several efforts have been made to perform Hough Transform faster. Li, Lavin and Le Master reported a Fast Hough Transform using a hierarchical approach [1]. The method they presented divides the parameter space into hypercubes. The feature points "vote" for these hypercubes and actual Hough Transform is performed only on the hypercubes exceeding a selected threshold. This algorithm is sequential, thus it is limited in the speed-up it can provide.

Most of the current work on improving the speed of the Hough Transform makes use of the Parallel processing techniques [25]. A literature survey about Parallel Hough Transform techniques shows that all the published research concentrates on a given architecture and gives results on that particular topology. Performance of the single algorithm on more than one topology is missing. Load balancing techniques are not mentioned or inadequate. Asynchronism overhead due to nonhomogeneous distribution of edge pixels in a binary image is a very important factor that decreases the performance of a parallel Hough Transform algorithm. Partial elimination of asynchronism overhead is reported by Ranka and Sahni [4]. Other published material lack the method to eliminate asynchronism overhead and some even claim that it is not possible to eliminate it without a priori knowledge about the distribution of pixels in the image [5].

This thesis develops two parallel Hough transform algorithms that eliminate asynchronism overhead and illustrates the results obtained on three different architectures which are linear array, mesh and hypercube. Elimination of asynchronism overhead in the algorithms do not require additional computational task, thus they are achieved by the very nature of the algorithms. Algorithms presented in this thesis accept binary image as input as against many other algorithms in literature that assume a predigested image with a linked list of all the "black pixels".

Our first "standard" algorithm separates the task of scanning the binary image from the task of generating the parameter space. One of the processors in the MIMD architecture is assigned the task of scanning the image, extracting the coordinates of the edge pixels and communicating these coordinates to the remaining processors. The original algorithm uses static partitioning of the image, i.e., it generates and communicates coordinate messages after scanning a fixed number of pixels. Our second "modified" algorithm uses dynamic partitioning of the binary image and adjusts the size of the current coordinate message according to the size of the previous coordinate message. It is shown that this modified algorithm is better than the standard algorithm in almost all cases.

1.2. Organization of the Thesis

Chapter two of this thesis covers the fundamentals of Parallel Processing and supplies detailed information about the operation of Hough Transform. Information about the experiment environment, NCUBE/10 hypercube computer is provided.

Chapter three, four and five are devoted to the implementation of the algorithms in a linear array, a mesh and a hypercube respectively. We used the same host machine, a large NCUBE/10 hypercube with

128 processors for all the implementations so that the architectural overheads are similar. The architecture sizes were varied from as few as three processors to as many as 31. The binary images used to characterize the performance of the algorithms had densities ranging from 0 to 25% and the distribution of black regions was also varied a great deal.

Chapter six summarizes the results obtained and possible future extensions.

Chapter 2

Hough Transform and Parallel Processing

2.1. Hough Transform

In many image understanding applications one has to recognize objects with predetermined shapes. Examples of such applications range from detection of missiles and planes for military purposes to the recognition of parts of specific shapes in an automated manufacturing situation.

Hough Transform (HT) is a well-known, and very efficient method for pattern recognition in digital images. Hough Transform dates back to a patent by P. V. C. Hough granted in 1962 [6]. Hough Transform can detect straight lines, circles and other curves that can be expressed through parametric equations. Recently Hough Transform has also been generalized to detect analytic curves and arbitrary shapes [7]. Hough transform can also be used to detect "filled in" or shaded objects. However, because the shape of an object is a property of its perimeter, in these cases one typically has to use an edge extracting operator to accentuate the object perimeter before one applies the actual Hough Transform.

Hough Transform has shown a good performance even in the presence of image noise and occlusion. This is a very important property in real applications since it is virtually impossible to get an uncorrupted image. Farther, removal of noise by say, low pass filtering, has a very adverse effect on edge detection since it smears the edges.

When using Hough Transform to detect straight lines in a binary image, one has to decide which parametrization will be used for the straight line. One of the options is, using slope-intercept parameters and expressing the line with the equation $y = mx + c$, where m is the slope and c is the intercept. The problem with this parametrization is that it leads to a parameter space boundless on both dimensions. It is possible to get around this problem by using the (ρ,θ) parameters of the normal where,

$$\rho = x\cos\theta + y\sin\theta. \tag{2.1}$$

This parametrization leads to a finite-sized parameter space. If one restricts θ to $[0, \pi)$, then it is possible to represent each straight line by its corresponding (ρ,θ) parameters uniquely.

Hough transform works by examining the image one pixel at a time and for each pixel that is black, determining all the lines that may contain that pixel. If the lines are characterized by their (ρ,θ) parameters, then a black pixel at (x_o,y_o) may be part of any line whose ρ and θ parameters are related by

$$\rho = x_o\cos\theta + y_o\sin\theta. \tag{2.2}$$

Thus each pixel in the image space generates a sinusoidal curve in the parameter space.

Corresponding to each black image pixel, the Hough transform increments the count of each (ρ,θ) cell that is on this sinusoidal curve in the parameter space. After the entire image is scanned, the Hough Transform examines the parameter space to identify the cell that has the highest count. This point corresponds to the line to which the maximum number of image points fitted.

As an example, consider 3 collinear points in the image space Figure 2.1.(a). As seen in Figure 2.1.(b), each of these three points generates a sinusoidal curve in the parameter space and they will intersect each other at a common point (ρ_o,θ_o). Returning to Figure 2.1.(a)., one recognizes that (ρ_o,θ_o) is the parameter pair for the line containing these three collinear points.

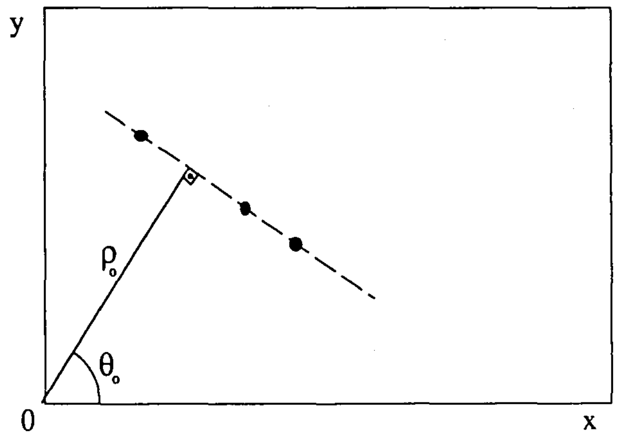

Figure 2.1.(a). 3 collinear points in the image space.

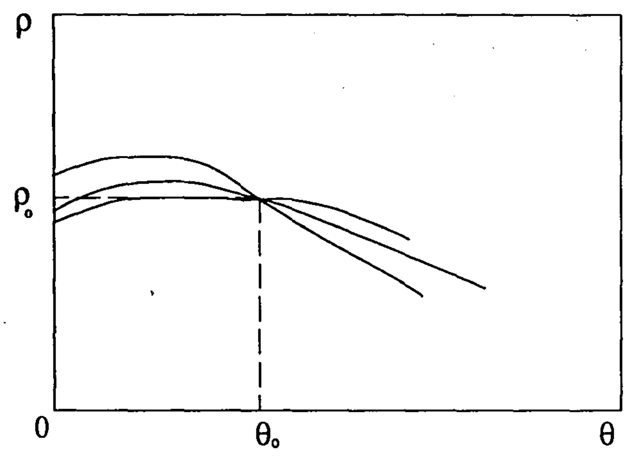

Figure 2.1.(b). Parameter space generated by 3 colinear points.

In practice, parameter space is quantized and basically consists of a 2-dimensional array in the case of straight line detection. As each point in the image is processed, (2.1) is used to find the radius values for each θ value and the corresponding cell count is incremented. After all the points in the image are processed, the parameter space is searched for cells exceeding a certain threshold value. Cells over the threshold value are picked up as parameters of the straight lines in the image. This is called the "Cluster detection". "Cluster detection" is complicated because in practical noisy images, straight lines tend to create clusters of high-count cells instead of stand-alone high-count singular cells. Depending on the application, various methods of cluster detection could be used once the parameter space is generated.

The generation of parameter space is extremely computation intensive. Assume that the image dimensions are NxN pixels, parameter space is MxM, and the average image density (ratio of black to total number of pixels) is ξ. One can see that this implies that one should examine N^2 image pixels, a task requiring $O(N^2)$ time. Farther for each of the ξN^2 black pixels one should update the parameter space by evaluating (2.1) for M different θ values. This task therefore requires a time $O(MN^2)$. Thus the complexity order of the parameter space generation is $O(MN^2)$. In real practice when the images have a typical size of 1024x1024 pixels and the θ parameter range is divided into,say, 90 slots, one can see that even for moderate image density of 5%, (2.1) will have to be evaluated for more than four million times for generation of a parameter space. One should also note that this estimate has to be farther increased because it does not include either the time required to scan the image or the time required to update the counts in the parameter space. Because of the enormous computational complexity, the problem of parameter space generation deserves new solutions. In this Thesis, we examine this problem in the light of parallel processing technology.

2.2. Various Applications of the Hough Transform

Hough Transform has many applications in industry, medical arena and the military. Depending on the requirements of the specific application, various modified versions of Hough Transform can be used. Although there are many of these modified Hough Transforms, one can group them into two broad groups.

The first group consists of modified Hough Transforms where accuracy of the transform is emphasized. These are generally used in medical applications or other delicate applications where precision has the top priority and computation time is not a big concern [8-11]. Most of the Hough Transforms in this group are generated by taking the nature of the problem into account and modifying the Hough Transform accordingly. After these modifications, the resultant Hough Transform could be very powerful for a specific problem but not very suitable for others. Thus, precision is increased at the expense of losing generality and increasing processing time. Other algorithms may use iterative methods to gain precision. This approach increases accuracy at the expense of increasing processing time dramatically. In general, complexity of the algorithms in this group is much greater than the complexity of the original Hough Transform algorithm.

The second group of modified Hough Transforms includes methods where reducing the time needed for Transformation is the most important goal. These methods try to compute Hough Transform with a pre-determined level of accuracy as fast as possible [1-4]. Most of these modified Hough Transforms make use of parallel processing techniques to achieve high performance. Usually, these algorithms leave the basic equation (2.1) unchanged and try to devise parallel algorithms for its evaluation. Here, one should note that some parallel algorithms may be specific to a parallel machine architecture. In designing these algorithms several factors should be taken account such as scalability, speed-up, efficiency, accuracy, etc. Applications for this group range from detecting military objects like missiles, tanks, etc. to robot guidance systems, industrial automation and quality control.

2.3. Parallel Processing

The ever-increasing need for computing power has forced rapid progress in parallel processing in recent years exploiting the advances in electronics technology. Parallel processing involves partitioning a computation into several concurrently executing tasks. A parallel computer consists of several processors connected together for purpose of jointly executing the parts of the same computational task. Parallel computer characteristics vary widely from each other. Number of processors can range from as few as four to several thousands. Parallel computers with processor numbers in the order of hundreds are usually called massively parallel computers.

In recent years, many commercial massively parallel computers have been introduced. By using thousands of microprocessors, massively parallel computers reach the supercomputer performance level with a much lower cost than a traditional supercomputer. Traditional supercomputers are designed to rely on very fast components and pipelined operations. As a result, the incremental performance improvement of these traditional supercomputers is increasingly expensive. On the other hand, performance of a massively parallel supercomputer can be increased by increasing the number of processing elements (nodes).

Parallel computers can be grouped into MIMD (Multiple instruction and Multiple data) or SIMD (Single instruction and Multiple data) machines. In MIMD computers, each processor asynchronously executes its own program. In SIMD computers, a central controller processor broadcasts the instructions to all the processors and forces them to synchronously execute identical sets of instructions. Node processors for MIMD machines tend to be more complex than the node processors for SIMD machines. MIMD machines support a variety of algorithms and multiple users, while the SIMD machines are typically single user machines that can execute a limited class of algorithms.

One other important factor in the design of parallel computers is the method of sharing results and data between the processors. One approach is to use global memory. In this scheme, each processor has access to a global memory space and data sharing is achieved through the global memory. Thus if processor A needs to send data to processor B, it writes it in the global memory and processor B retrieves the data by reading the memory This scheme solves the problem of sharing data between the processors but creates a new memory contention problem. Memory contention occurs when two or more processors try to use the same memory location at the same time and may result in serious performance loss.

On the other hand, in the distributed memory systems each processor node has its own local memory holding local variables. Data sharing is achieved by sending messages between the nodes. In such message passing architectures, the connection network between processors is a very important design criterion of the parallel computer. Although it is desirable to provide communication links from each processor to every other, this is usually not possible because of practical limits on the number of connections. Therefore each processor is directly connected to only a few other processors and messages to the rest are routed through these intermediate nodes. Some of the widely used interconnections are mesh, ring, binary tree and hypercube. Amongst these, the hypercube is the most popular because by using its few selective links, it is possible to simulate binary trees, linear arrays and meshes in hypercubes. The hypercube parallel computers are discussed in greater detail in the next section.

2.4. Hypercube Parallel Computers

A hypercube is a collection of processors interconnected by a communication network to provide a good performance/cost ratio. An n-dimentional hypercube can be recursively defined by duplicating the (n-1)-dimensional hypercube and connecting the corresponding nodes. (A degree 0 hypercube is a single processor). Thus an n-dimensional hypercube will have 2^n nodes and each node is connected to its n neighbors. By using the recursive definition, one can label each node of an n-dimensional hypercube by a unique binary n-tuple such that nodes whose n-tuples differ in only one bit are neighbors. A few hypercube examples are shown in Figure 2.2.

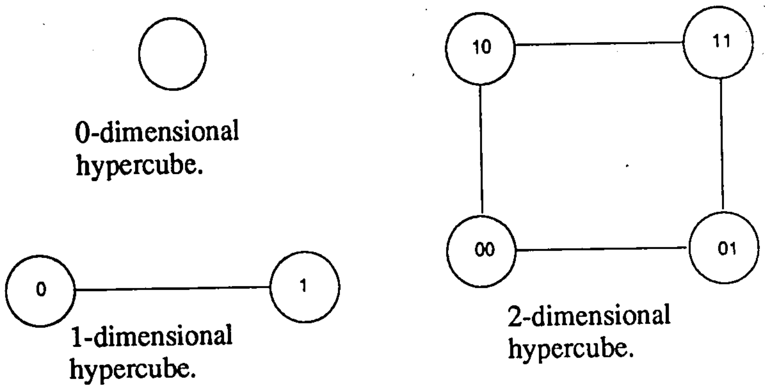

Figure 2.2.(a). Hypercubes of degree 0,1 and 2.

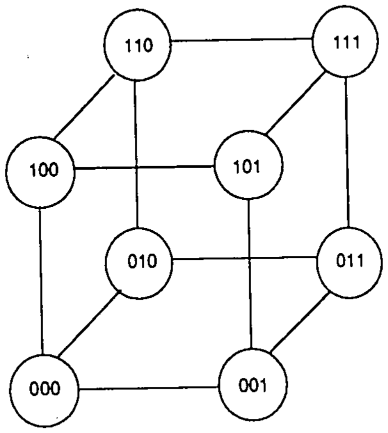

Figure 2.2.(b). Hypercube of degree 3.

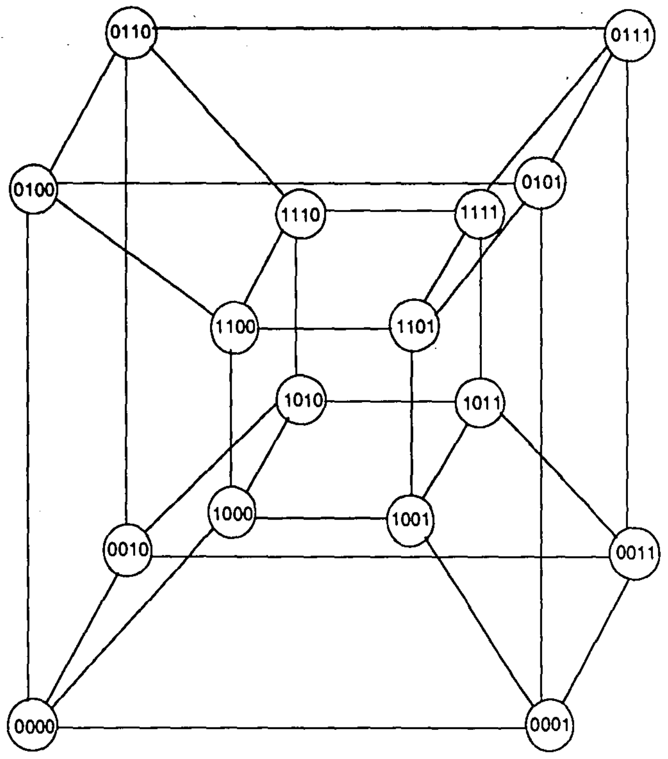

Figure 2.2.(c). Hypercube of degree 4.

Hypercube topology has many advantages. It is homogeneous in the sense that all nodes are identical. In an n-dimensional hypercube, each node has n links to manage. So the total number of links is equal to nN/2 , where $N = 2^n$ represents the total number of nodes.

Diameter is the maximum distance between any nodes of a network which sets an upper bound on the maximum message communication time. Diameter of hypercube topology with 2^n processors is just n.

In most of the hypercubes, each node has its own memory and communicates with other nodes by passing messages. In 1983, the 64-node Cosmic Cube at Caltech became the first working hypercube computer [12]. Since then many commercial versions of hypercube have become available.

One of the consequences of the recursive hypercube topology is that it can be partitioned into smaller degree subcubes, which are hypercubes themselves. Thus a hypercube can be easily used as a multiuser system. Most operating systems in commercial hypercubes use this feature and can simultaneously allocate, for example, a degree 6 hypercube to one user, and degree 5 hypercubes to two users from an available degree 7 hypercube.

Parallel algorithm implementations are evaluated with the help of two performance criteria, speedup and efficiency. Suppose one has a parallel algorithm that uses p processors. Let T_s be the optimal serial time to solve a problem and T_p be the time required to solve the same problem using p processors. Then speedup, SP(p) and efficiency, E(p) of this parallel algorithm are defined as,

$$SP(p) = T_s / T_p \tag{2.3}$$

$$E(p) = SP(p) / p \tag{2.4}$$

Speedup describes the speed advantage of the parallel algorithm over the best serial algorithm. Ideally, $SP(p) = p$. Efficiency is a measure of what fraction of ideal speed-up has been achieved. Note that $E(p) < 1$ in practice because of the overheads involved in parallel algorithms.

2.5. NCUBE/10 Parallel Computer

NCUBE/10 is a commercial MIMD hypercube manufactured by NCUBE Corporation. A fully configured NCUBE/10 system can accommodate up to 1024 nodes, each based on a 32-bit custom processor. Each processor has a peak performance of 0.5 MFLOPS, so that fully configured system has a potential throughput of approximately 500 MFLOPS. Each node processor has 11 bidirectional DMA channels. 10 of these 11 channels are used for connection to other nodes and remaining 1 is used for connection to an I/O board. 64 processors, each with a RAM size of 128K form a 16 x 22-inch board. There is no global memory in NCUBE/10. In the full configuration, 16 of these boards are connected to 16 front-end processors (connected as a hypercube themselves) for communication between boards. These front end processors also act as the "host" processors to provide I/O capabilities through a multiuser Unix-based operating system called AXIS.

Communication between the processors is managed by sending messages. Augmented C and FORTRAN compilers allow the programmer to send messages between the nodes with the help of additional functions. Some of special C functions are "nopen()", "nloadm()", "whoami()", "nwrite()", "nread()" and "ntime()".

"nopen(n)" is used in the host program and it allocates a subcube of degree n and returns a channel number which is used as a reference number for the subcube allocated. "nloadm()" is used in the host program to load node programs to the specified nodes. "whoami()" is used in the node program and it returns node number of the calling processor and id number for communicating with the host. "nwrite()" sends the message along with its length and type to the specified node or to the host. "nread()" accepts the next available message with the suitable source and type. "nread()" is a blocking function, so it doesn't return the control until the message is received. "nread()" and "nwrite()" functions are used both in node programs and host programs. The time since the node was initialized is returned by function "ntime()". To convert the "ticks" returned by "ntime()" into seconds, one needs to multiply it with (1024/N) where N is the clock frequency.

The programs and data is loaded into the nodes using the host processors. Once this is accomplished, each node works on its own data independently. Thus host does not have control over the nodes once they start working and there is no global clock to synchronize the nodes. This asynchronous operation actually provides more flexibility because if needed, the task execution can be synchronized by appropriate message passing.

The NCUBE/10 at Lehigh does not allow the nodes to communicate directly with external peripherals such as the file system, terminal, etc. Thus all such interactions have to go through the host processor.

Users of NCUBE/10 can open subcubes of various sizes at the same time, thus it has a high degree of flexibility. Subcubes of the main hypercube are completely isolated from one another. To provide communication between the nodes, each of the NCUBE/10 nodes contains a small monitor system called VERTEX.

2.6. Communication in NCUBE/10

Time required to pass B bytes to a neighbor node, t_c , is given by (2.5) in NCUBE/10.

$$t_c = \alpha B + \beta \tag{2.5}$$

In (2.5), α represents the rate of transfer and β represents the communication set-up time. Our experiments showed that $\alpha = 0.017$ [ticks / bytes] and $\beta = 4$ ticks. As an example, sending 2000 bytes from node 0 to node 1 will be completed in 38 ticks. A tick corresponds to 1024 clock cycles and for a 6MHz system NCUBE/10, it is approximately 0.171 msecs.

2.7. Binary Images Used

Images with size of 64x64 pixels were used in this work. These binary images were described by mainly two parameters, the image density and the image distribution. Image density of a binary image is defined as,

$$\text{Image density} = (\text{ \# of black pixels}) / (\text{\# of total pixels}) \tag{2.6}$$

Image density is a measure of the blackness of the image. Generally this translates to the number of edges contained in the image. The image densities examined in this thesis virtually cover the entire range of practical interest. Note that the amount of computation required increases linearly with the number of black pixels, or the image density.

Another factor related to the digital images is the distribution of the black pixels in the binary image. Since the image distribution can be varied in a multitude of continuous ways, we limited our experimentation to a somewhat preconstrained setting. We let the image be divided vertically in two equal halves with uniform densities in each half. We then varied the distribution of black pixels between the two halves. The images with 10% image distribution have 10% of their total number of black pixels in the first half of the image and 90% of their total number of black pixels in the second half of the image. For each of the images considered, image distribution was varied from 0% to 100%. 0% image distribution implies all the black pixels in the second half of the image and 100% image distribution means that all the black pixels are in the first half of the image. The distribution of black pixels are taken into account with the criterion described above rather than, say, variance criterion, because examining the image always start from one end of the image and it is important to know if the black pixels are encountered at the beginning of the image or at the end of the image.

Chapter 3

Implementations on a Linear Array

3.1. Introduction

In this Chapter, the results of Hough Transform execution on a linear array are presented. As mentioned earlier, it is possible map a linear array onto a hypercube topology. The results presented here are obtained by mapping linear arrays of different lengths on hypercube NCUBE/10.

A linear processor array consists of n processors numbered 0,1, ...,(n-2),(n-1) with a bidirectional communication link between every pair of successive processors. Thus node i, where $0 < i < (n-1)$, is connected to nodes (i-1) and (i+1). The maximum distance between any pair of processors of the linear array is (n-1). Figure 3.1. shows a linear array with n processors.

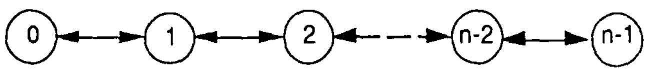

Figure 3.1. Linear Array with n processors.

Linear Arrays draw considerable interest because implementation of parallel algorithms is relatively simple and uniform on them. Even though there are no references to the execution of Hough Transforms on Linear Array, they have been used to host a variety of parallel algorithms in the past [13-16].

3.2. Embedding a Linear Array in Hypercube

A linear array may be mapped onto a hypercube in such a manner that the connected processors of linear array get assigned connected processors of a hypercube. Although a multitude of such mappings are possible, it is desirable to use a systematic approach to this mapping problem. One such systematic method is called the gray code (or reflected gray code) mapping.

Gray code mapping is generated as follows: assume that one has a hypercube of degree m, then one will have $n=2^m$ nodes. Each node will have an m bit label, p, where $0 \leq p \leq (n-1)$. Let each processor in the linear array be represented by another m bit string,s.

Thus p is the "host" processor number (in the hypercube) and s is the "guest" processor number (in the linear array). Define function g as follows;

$$g(s) = s \quad s^{\wedge}, \qquad (3.1)$$

where, denotes an ExOR operation and $s^{\wedge}$ stands for $\lfloor s/2 \rfloor$. In general $s^{\wedge n}$ represents ($s^{\wedge(n-1)}$)$^{\wedge}$, i.e., n applications of the operation $^{\wedge}$. One can easily show that g(s) can be used as the host processor number p because g(s) g(s+1) is a vector of weight 1. Thus the processors to which linear processor s and s+1 map are connected in the host (as demanded by the linear array geometry).

(3.2) gives the expression for g^{-1}, inverse function of g, which can be used to obtain s, given p.

$$s = g^{-1}(p) = p \;\; p^{\wedge} \;\; p^{\wedge 2} \ldots \; p^{\wedge(m-1)} . \qquad (3.2)$$

(3.1) and (3.2) define a gray code mapping and give a direct and easy method to map linear array onto a hypercube. Note that this does not restrict the size of the linear array. If the total number of processors in the linear array is s_total, one can always map it onto a hypercube of degree $\lceil \log_2 \text{s\_total} \rceil$.

A sample gray code mapping of a size 8 linear array on a degree 3 hypercube is shown in Table 3.1. The s and p values in Table 3.1. satisfy (3.1) and (3.2). One can verify from this table that the processors directly connected in the linear array are also directly connected in the hypercube.

Table 3.1. 8 node Linear Array mapping for a hypercube of degree 3 using Gray code.

s	p	s (decimal)	p(decimal)
000	000	0	0
001	001	1	1
010	011	2	3
011	010	3	2
100	110	4	6
101	111	5	7
110	101	6	5
111	100	7	4

3.3. Implementation

One of the advantages of the Gray Code mapping is its algebraic expression. It is possible for each node to compute other nodes directly connected using this expression. To illustrate this, consider that node number p_o, uses variable next_proc for the node number of the next processor and uses prev_proc for the node number of the previous processor. In this case, next_proc and prev_proc could easily be calculated with (3.3) and (3.4) as

$$\text{next\_proc} = g(\ g^{-1}(p_o) + 1\ ), \tag{3.3}$$

$$\text{prev\_proc} = g(\ g^{-1}(p_o) - 1\ ). \tag{3.4}$$

As explained in Chapter 2, the NCUBE/10 requires message source or destination node numbers in nwrite() and nread() statements. At the beginning of the node program, each node can get its node number p, using the command whoami() and then using (3.3) and (3.4), it can compute the node numbers of its neighbors.

Most of the Parallel Hough Transform algorithms discussed in the literature assume that the coordinates of the pixels are provided to the processors. In real life situations, one has to process the original binary image and obtain the coordinates the black pixels. This is a task that should be taken into account when designing the algorithm. In fact, for a large image size this task could be very time consuming. In the analysis throughout this Thesis, the time required for extracting the coordinates of the black pixels is considered as part of the total execution time.

A binary image is obtained by applying an edge extracting operator to original digital image. Black pixels in a binary image tend to be distributed very unevenly because there may be lots of edges in one part of the image while another part may have none or few edges. This unevenness in distribution of black pixels causes asynchronism overhead in parallel Hough Transform algorithms which use the method of distributing image space over the processor nodes. In the algorithms used in this Thesis, obtaining the the coordinates of the black pixels from the binary image and generating the Hough Transform parameter space is handled independently.

In linear array algorithms, processor 0 examines the binary image and extracts the coordinates of the black pixels. It creates a message containing the entire array of these pixels and then sends it through the array touching processors 1,2, ...,(n-1). These processors are expected to generate parameter space split equally between them. In our first algorithm, (called the "standard algorithm"), processor 0 generates a message after examining 1/8 of the image. Thus in this algorithm, which follows static partitioning, the message generation is independent of the image data and each processor receives exactly eight coordinate messages.

In our second algorithm, (called the "modified algorithm"), the message generation times are unknown. The modified algorithm adjusts the time current message will be sent according to the size of the last message. In general, a new message is generated at a time so as to arrive at the next processor just as it finishes working with the data in the old message. This dynamic message generation eliminates idle time of processors 1 through (n-1).

Figure 3.2. shows the allocation structure in the case of (M+1) total processors. The structure is same for both standard and modified algorithms while the algorithm for generating the messages is different. In Figure 3.2., the direction of the arrows shows the direction the coordinate messages travel.

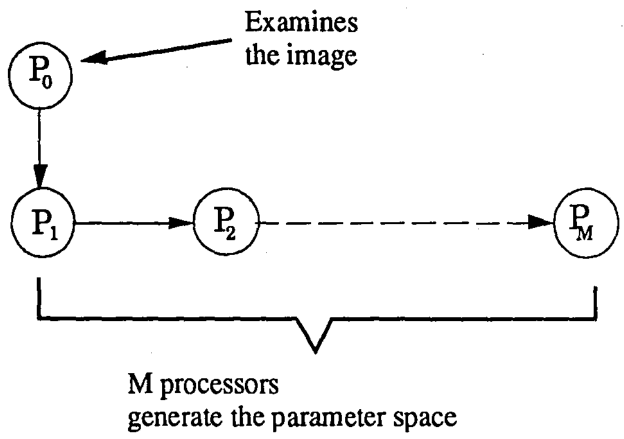

Figure 3.2. Allocation of processors used in Linear Array.

In the actual implementation of Linear Array Hough Transform algorithms on NCUBE/10, the total number of processors used were 4, 7, 11, 16 or 31. In each case, one processor was allocated to examining image and the rest were assigned to the task of generating parameter space. The mapping of the chosen linear arrays on the NCUBE/10 is shown in Figure 3.3.

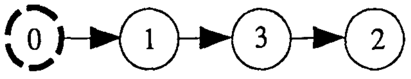

Figure 3.3.(a). Implementation of 4 processor Linear Array.

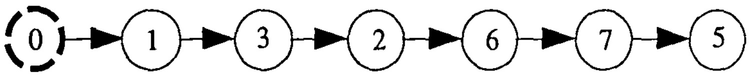

Figure 3.3.(b). Implementation of 7 processor Linear Array.

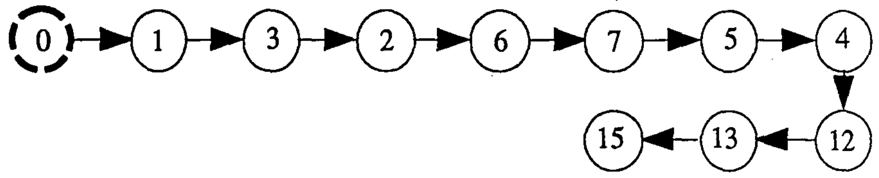

Figure 3.3.(c). Implementation of 11 processor Linear Array.

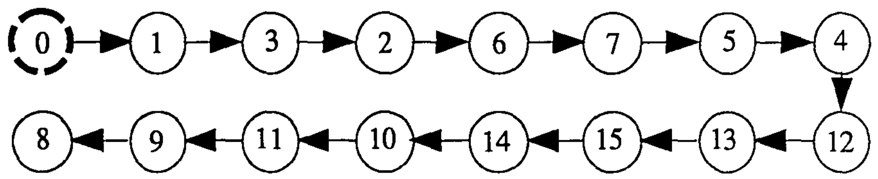

Figure 3.3.(d). Implementation of 16 processor Linear Array.

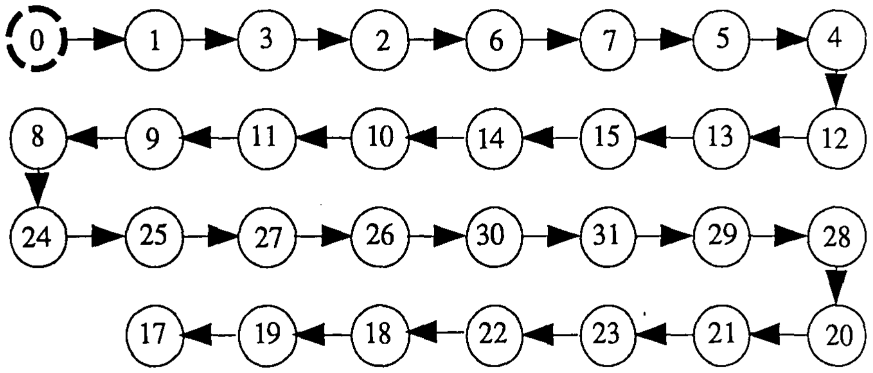

Figure 3.3.(e). Implementation of 31 processor Linear Array.

As discussed earlier, the range for parameter θ is $[0,\pi)$. This range was quantized into 90 slots and was distributed among the generator processors so that each processor has (90/M) slots where (M+1) was the total number of processors in the linear array. Each slot corresponds to 2 degrees or 0.0349 radians. For example, for 31 processor linear array, one processor examined the image and 30 processors updated parameter space, each for their assigned 3 angle slots.

3.4. Results

Figure 3.4.(a). gives the times obtained from Linear Array standard algorithm working with 31 processors. Figure 3.4.(b). is the corresponding graph for Linear Array modified algorithm. The percentage numbers on the lines indicates the image density of the image processed.

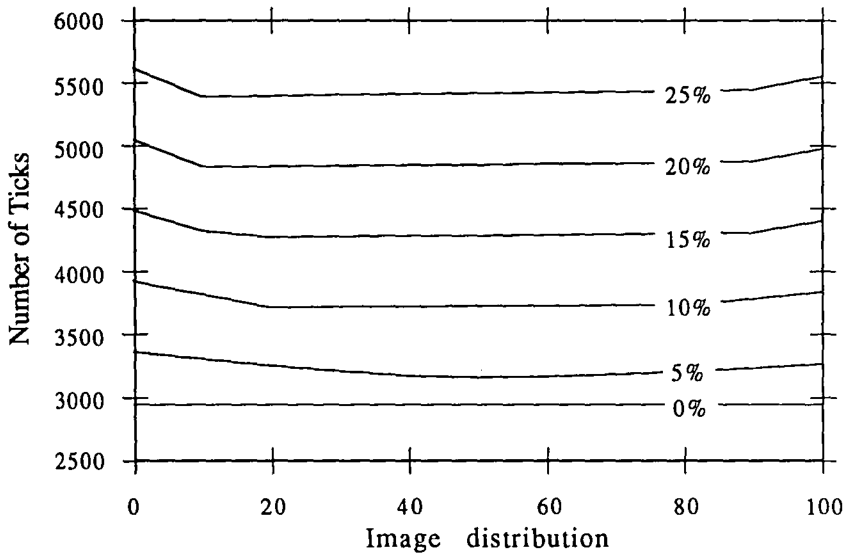

Figure 3.4.(a). Dependence of Time on image distribution Standard Linear Array / 31 processors.

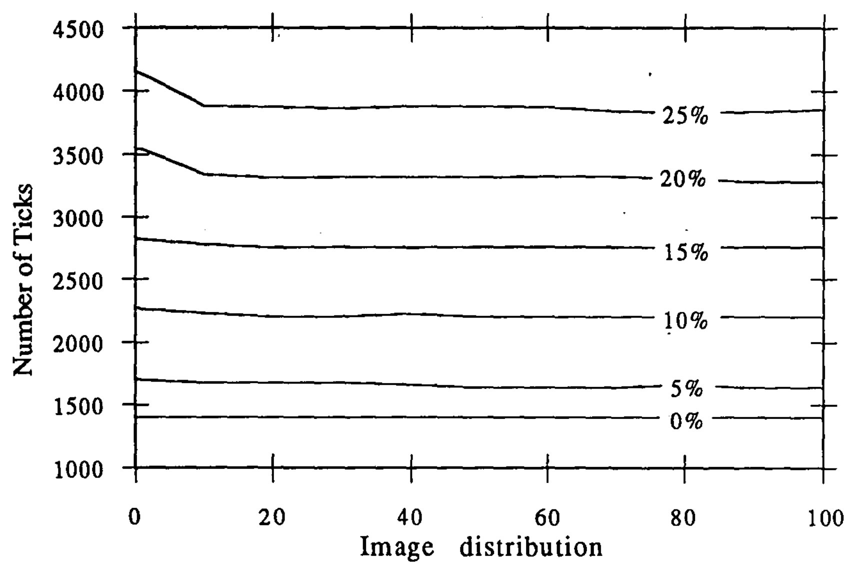

Figure 3.4.(b). Dependence of Time on image distribution Modified Linear Array / 31 processors.

From the results of Figure 3.4., one can easily see that the curves are almost flat for each case. This means that the number of ticks does not vary greatly with varying image distribution. This is a very desirable quality in an algorithm because it implies that the image distribution does not affect the performance of our two algorithms. Another even more important fact is that dependence on distribution would have suggested inefficiency because higher times obtained for oddly distributed images could be traced to asynchronism overheads. Figure 3.4. shows that even the standard algorithm eliminates the asynchronism overheads while modified algorithm improves the performance of the standard algorithm by reducing the total time and also by making it even less dependent on the image distribution. The experiments carried out with linear arrays of other sizes, namely 16, 11, 7, 4, also showed that the total time required to complete Hough Transform is independent of image distribution. Thus, by showing that both of the algorithms presented are independent of image distribution, the other related data are given based on random binary image at the particular image density.

As explained in Chapter 2, Speed-up and Efficiency are two of the important criteria used in parallel algorithm performance evaluations. Figure 3.5.(a). and Figure 3.5.(b). give the speed-up numbers attained in both Linear standard and Linear modified algorithms.

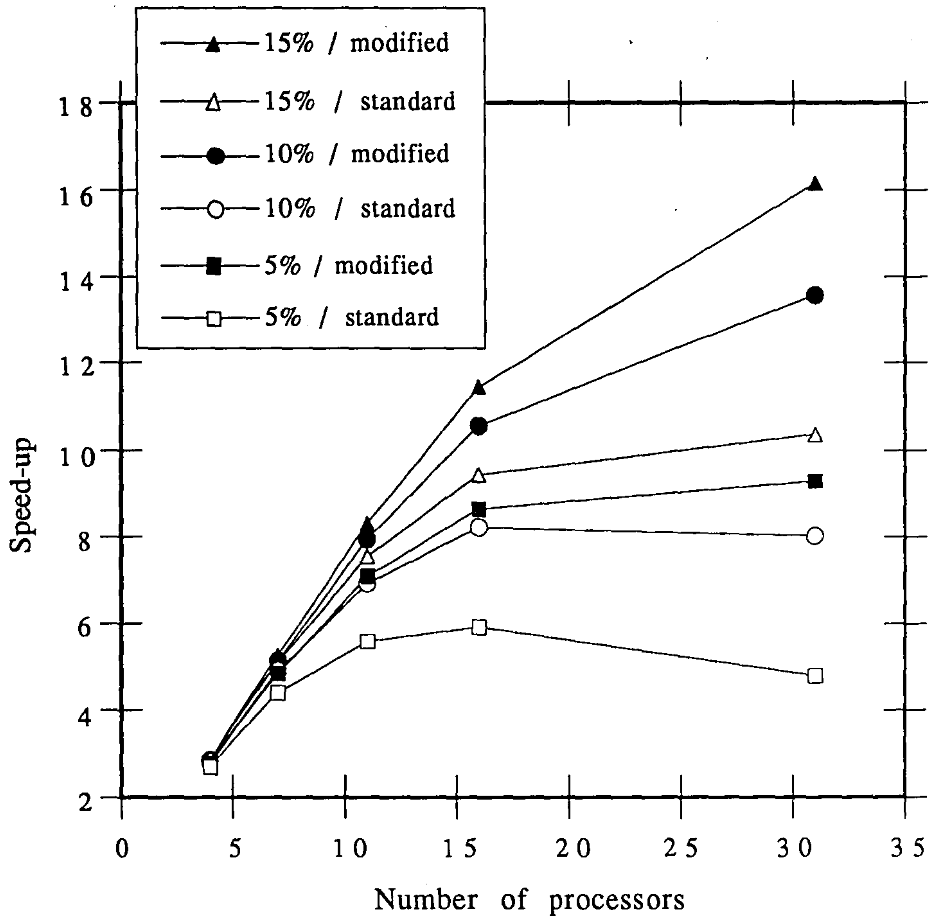

Figure 3.5.(a). Dependence of Speed-up on number of processors. Image densities: 5%, 10%, 15%.

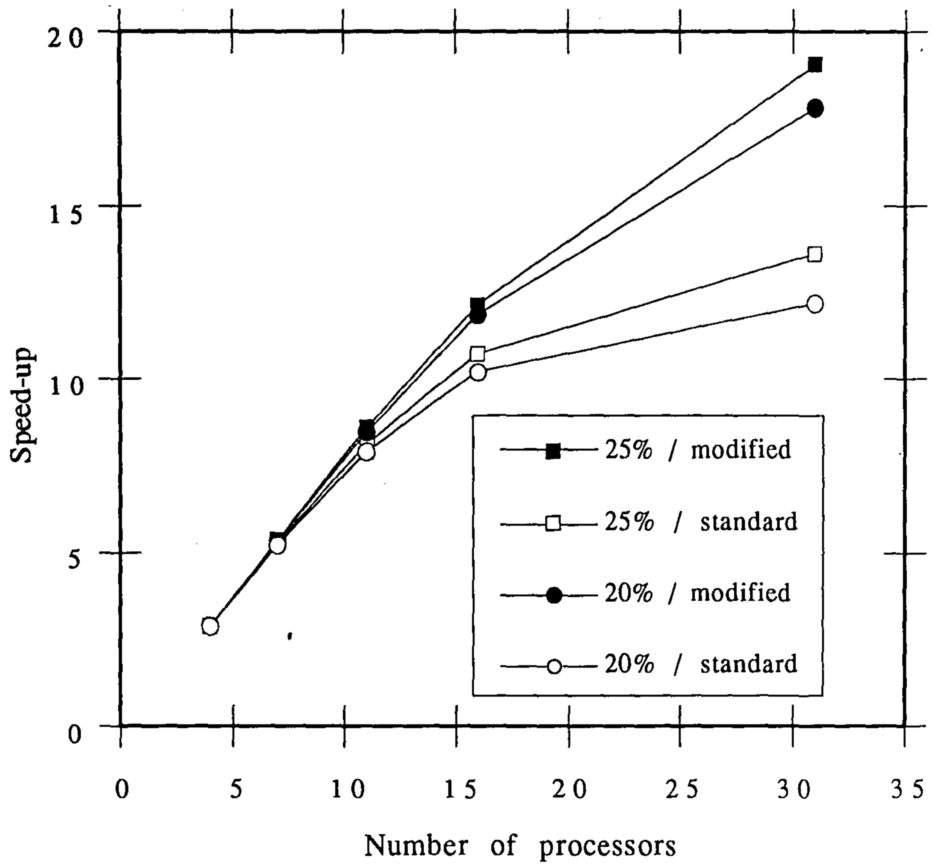

Figure 3.5.(b). Dependence of Speed-up on Number of Processors. Image densities 20%, 25%.

Examining Figure 3.5.(a). and Figure 3.5.(b). , one can immediately realize that the speed-up attained by Linear Modified algorithm is much better that the Linear Standard algorithm in every case presented. It is important to note that Linear Modified Algorithm outperforms Standard Algorithm in all linear array sizes and all image density values possible. This stems from the fact the modified algorithm adjusts the size and sending of coordinate messages intelligently to utilize the processing power of the calculating nodes.

Table 3.2. gives the improvements achieved by using the modified algorithm instead of using the standard algorithm.

Table 3.2. Percent improvement in speed-up due to Modified Algorithm.

Image density / No. of processors	5%	10%	15%	20%	25%
4	2.9	1.2	0.7	0.5	0.3
7	10.4	5.0	3.2	2.1	1.6
11	26.5	14.5	10.0	7.2	5.7
16	45.5	28.9	21.5	16.3	13.0
31	93.1	69.3	55.8	46.4	40.0

The efficiency of the Linear Modified Array as image density changes is illustrated in Figure 3.6.

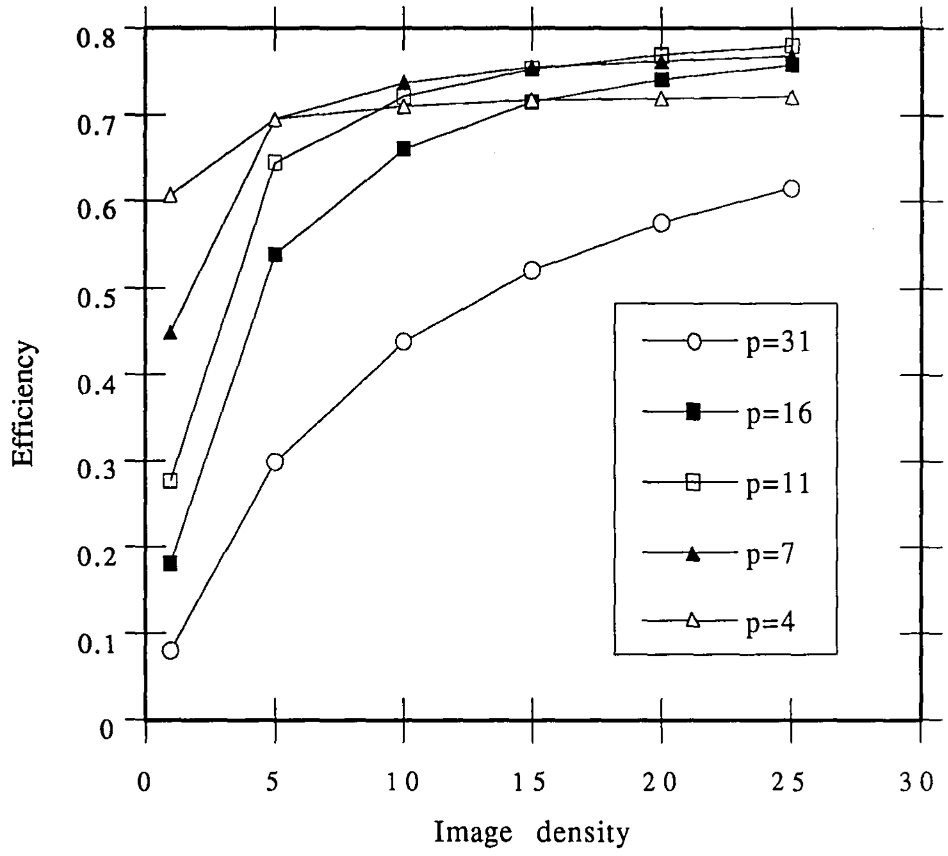

Figure 3.6. Dependence of efficiency on Image density Modified Linear Array Algorithm.

It is seen from Figure 3.6. that efficiency increases linearly with small image density and saturates after a while. This saturation value of the efficiency is dependent upon the size of the array. As one can see from Figure 3.6. efficiency of 4 processor Linear array saturates at 5% image density to about 0.7. As seen from Figure 3.6., efficiency of 31 processor Linear array is lower than the others and keeps on increasing in the range of image densities considered. On the other hand, a 16 processor linear array performance saturates at image density of 15% to a value of 0.75.

Chapter 4

Implementations on a Mesh

4.1. Introduction

In this Chapter, the results of Hough Transform execution on a two dimensional mesh are presented. The results presented here are obtained from meshes of various sizes mapped on hypercube NCUBE/10.

An n-processor mesh consists of $n=N^2$ processors arranged on a square grid of size NxN. There is a direct communication link between nearest neighbors. Each processor, including the processors on the borders and edges, has 4 links to manage with its neighbors. The number of links each processor has to manage does not depend on the total number of processors present. Figure 4.1. shows a mesh with n processors.

Meshes can have either wraparound or non-wraparound communication links. Diameter of a wraparound mesh is roughly half the diameter of non-wraparound mesh. Thus, wraparound mesh has better communication performance than a non-wraparound mesh. For example, processor (0,0) is connected to processors (0,1), (1,0), (0,N) and (N,0) in a wraparound mesh while it is only connected to (0,1) and (1,0) in a non-wraparound mesh. The meshes considered in this Chapter are all wraparound meshes.

A wraparound mesh of size NxN has a diameter of N when N is even and (N-1) when N is odd. Comparing a mesh with linear array, one can easily see that mesh has better connectivity than linear array.

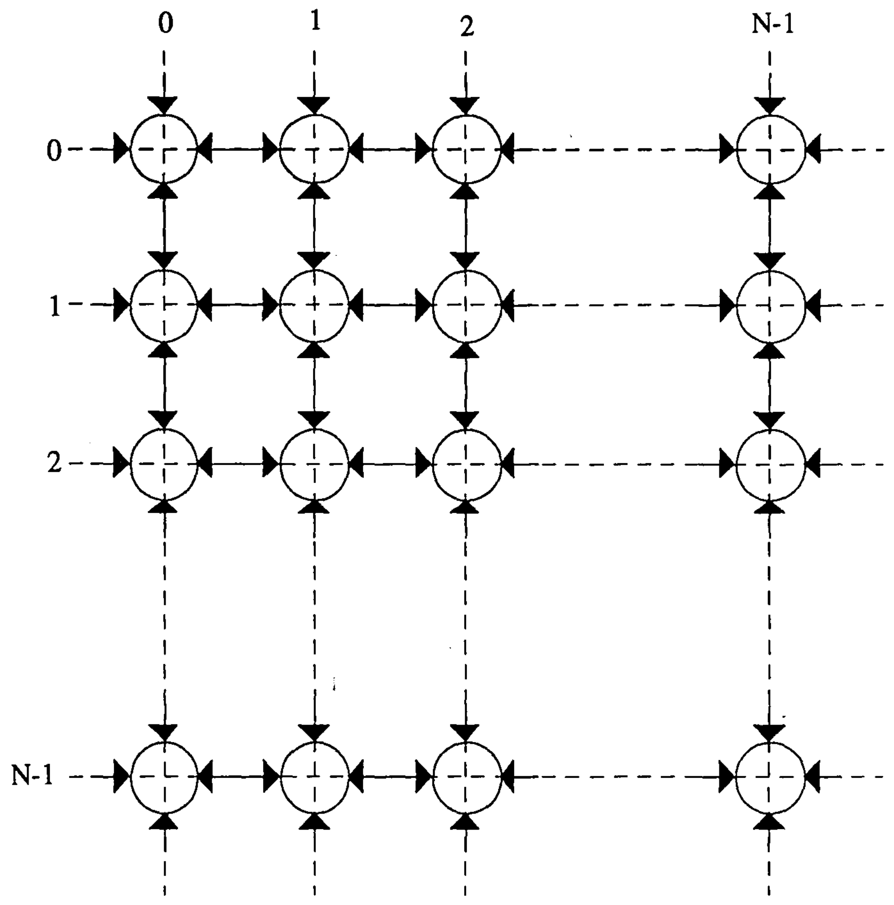

Figure 4.1. Mesh with NxN processors.

Mesh is one of the most popular architectures used for implementing parallel algorithms. There are many articles published about meshes used in parallel processing [17-19]. Meshes are especially suitable for problems closely tied to the geometry of physical space. For example, meshes are widely used in parallel image processing algorithms because mapping an image onto a mesh is relatively simple.

Kannan and Chuang [20] have reported a Fast Hough Transform implementation using meshes. The data from the actual implementation is not given in their paper. Their algorithm has two versions. First version is for the case when coordinates of the edge pixels are provided as input. As explained earlier, this approach does not take into account the time required for extracting the coordinates from image. Their second version of the algorithm accepts the actual image as input but this version is liable to asynchronism overhead because processors hitting the blank parts of the image remains "idle".

4.2. Embedding a Mesh in Hypercube

It is possible to embed a mesh with NxN processors in a hypercube of degree d, where $N=2^{d/2}$ and d is even. In this case, "host" processor number in the hypercube, p, is a d bit string. First d/2 bits of p are used to find the row number and the last d/2 bits of p determine the column number of the processor in the mesh according to the gray code ordering. Processor 0 is at row 0 and column 0. Embedding a 16 node mesh in hypercube of degree 4 is given in Table 4.1.

Table 4.1. Embedding 16 node mesh array in hypercube of degree 4.

Row Number \ Column Number	0	1	2	3
0	0000	0001	0011	0010
1	0100	0101	0111	0110
2	1100	1101	1111	1110
3	1000	1001	1011	1010

4.3. Implementation

Standard and modified algorithms were implemented in meshes with 4, 7, 11, 16 and 31 processors. The basic structure of the algorithms was the same as the linear array algorithms. Processor 0 examines the binary image and extracts the coordinates of the black pixels and remaining processors generate parameter space split equally between them. The main distinction of the mesh algorithm from the linear algorithm is that each processor passes the coordinate message to a list of processors instead of passing it to only one processor. Thus mesh algorithms are more general than linear algorithms. In linear algorithms the list of destination processors was not used because there was only one destination processor in each case and using the list approach reduces the efficiency unnecessarily in this case. The actual allocation of processors is shown in Figure 4.2. The dashed circle denotes the processor examining the image, the regular circle denotes the parameter space generator processors. The direction of the arrows indicate the direction the coordinate messages travel. The numbers in the circles indicate, p, the "host" processor number in the hypercube.

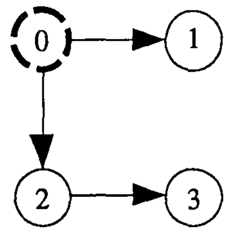

Figure 4.2.(a). Implementation of the algorithm with 4 processors.

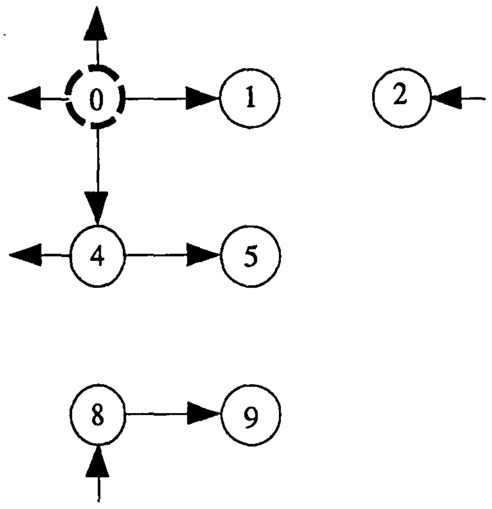

Figure 4.2.(b). Implementation of the algorithm with 7 processors.

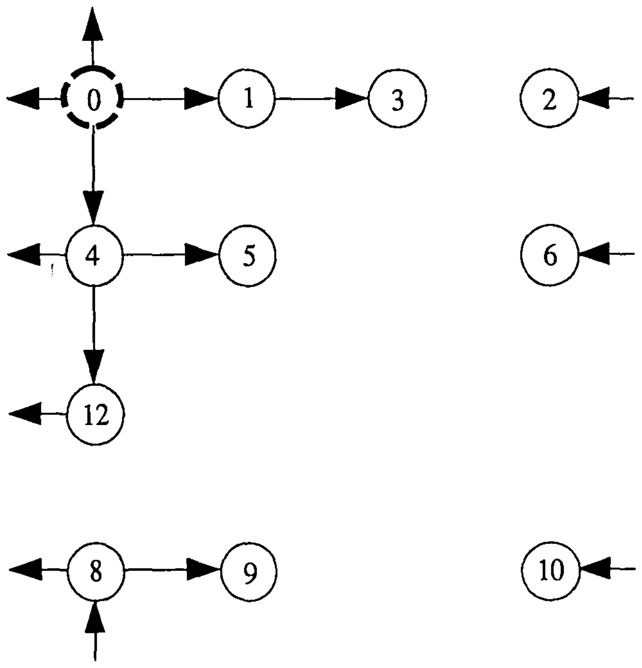

Figure 4.2.(c). Implementation of the algorithm with 11 processors.

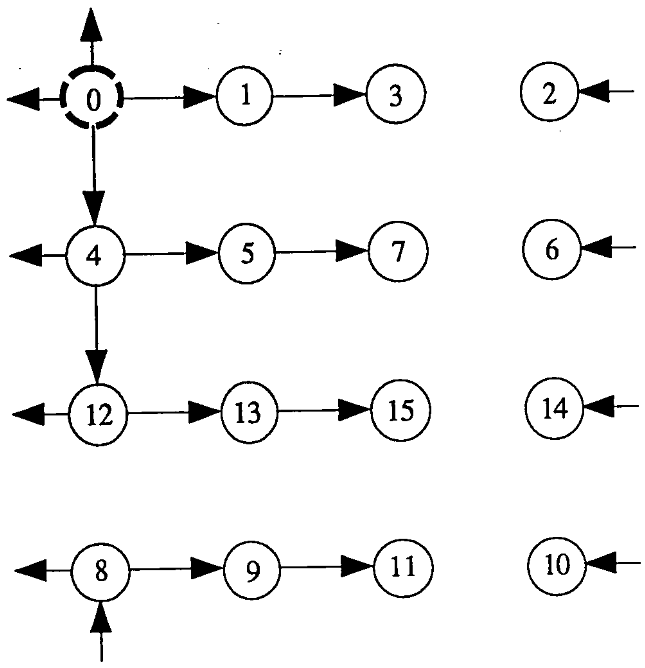

Figure 4.2.(d). Implementation of the algorithm with 16 processors.

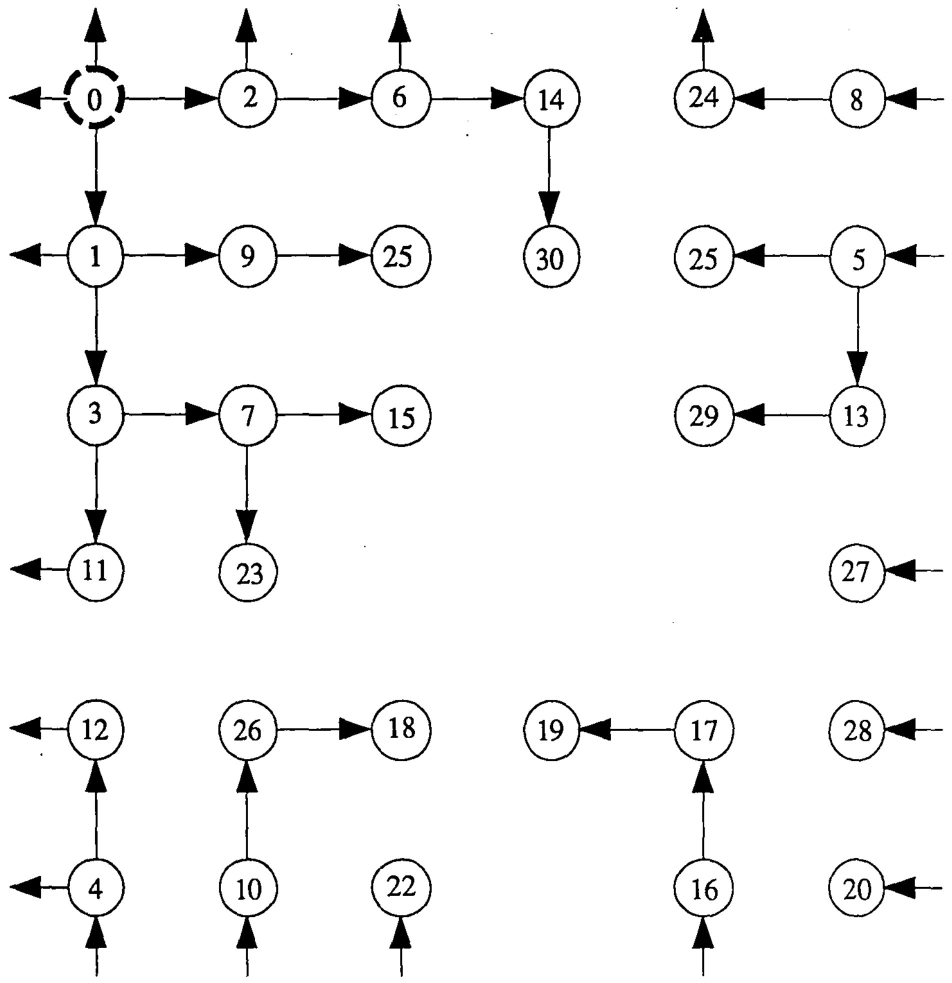

Figure 4.2.(e). Implementation of the algorithm with 31 processors.

Unlike linear arrays, it is possible to arrange communication paths in various ways in meshes. The communication paths in the algorithms used here are arranged so that no node gets the same message twice from different sources. The paths are also designed to minimize the longest distance from processor 0. Minimizing the longest distance is very important because the processor that is most distant from processor 0 receives the coordinate messages last and it is the last one to complete its task. Hence minimizing the longest distance in a given architecture increases efficiency.

4.4. Results

Figure 4.3.(a). gives the times obtained from mesh standard algorithm working with 31 processors. Figure 4.3.(b) is the corresponding graph for mesh modified algorithm. Image density of the binary image processed is indicated by the percentage numbers on the lines.

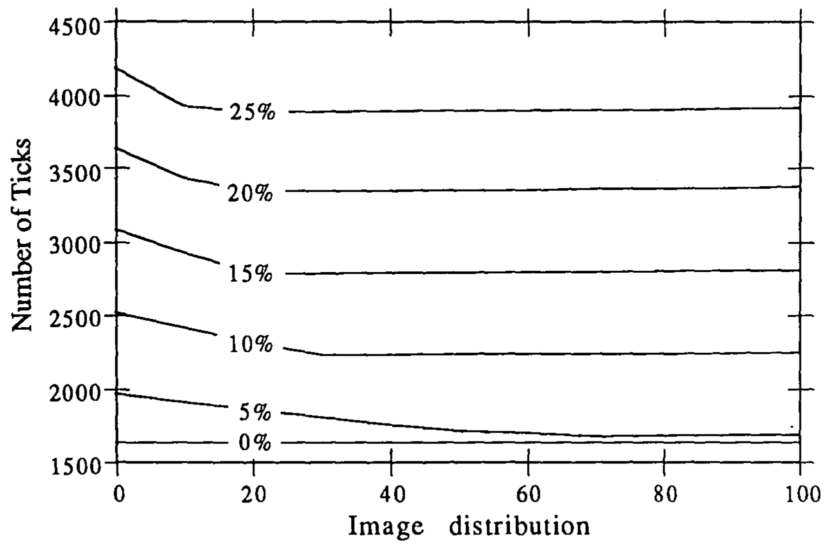

Figure 4.3.(a). Dependence of Time on image distribution Standard Mesh / 31 processors.

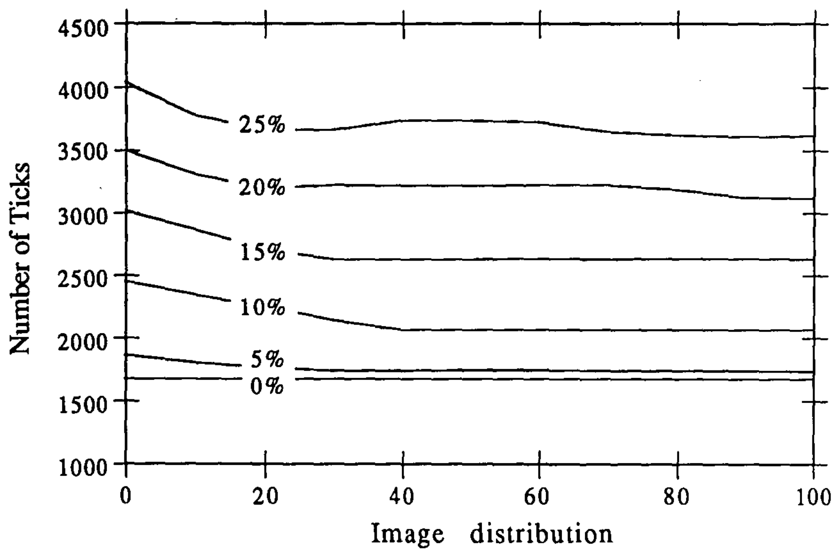

Figure 4.3.(b). Dependence of Time on image distribution
Modified Mesh / 31 processors.

Figure 4.3. reveals that the curves are almost flat for each image density shown. Thus, similar to the linear array, the image distribution does not affect the total time required to complete the generation of parameter space. As discussed earlier, this is a desirable property and indicates that algorithm is not liable to asynchronism overheads resulting from image distribution. Results obtained from programs using meshes with 4, 7, 11, 16 processors also exhibited the independence from image distribution. Because the times obtained are independent from image distribution the result of other experiments are given based only on random binary image at the particular image density.

Figure 4.4.(a). and Figure 4.4.(b). give the speed-up attained by standard and modified algorithms in meshes.

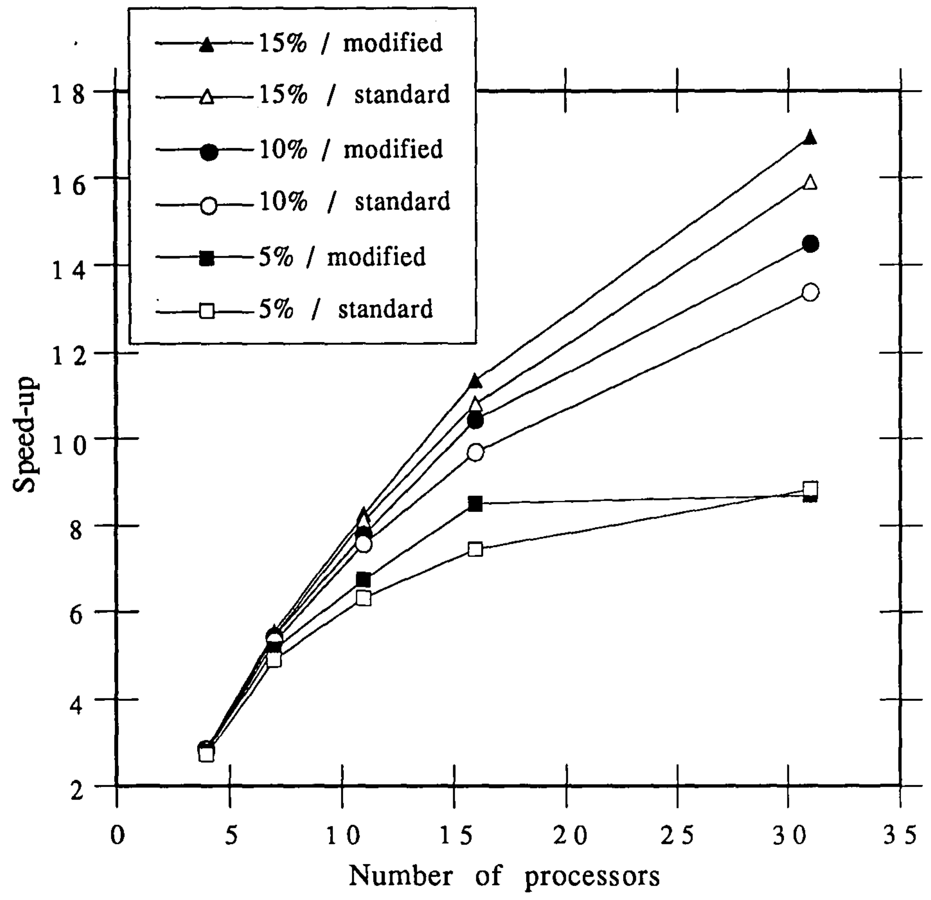

Figure 4.4.(a). Dependence of Speed-up on number of processors for image densities 5%, 10%, 15%.

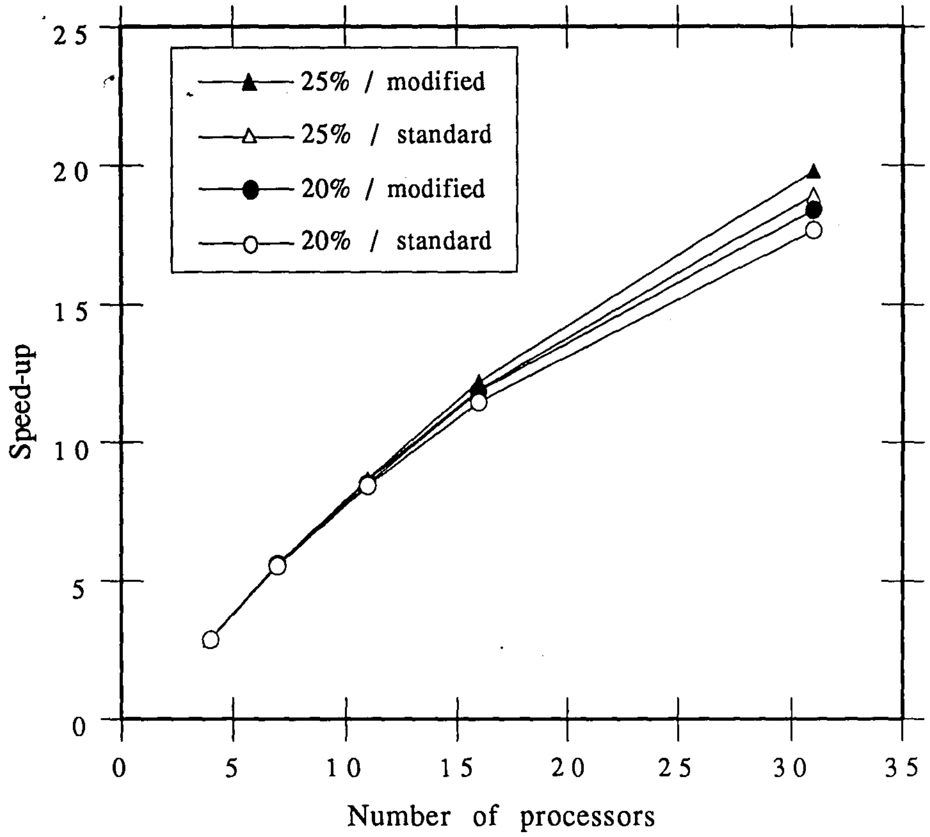

Figure 4.4.(b). Dependence of Speed-up on number of processors for image densities 20%, 25%.

Examining Figure 4.4. one can notice that the speed-up values increase as number of processors increase. This is a desirable quality because it means that one can decrease the total execution time simply by using more processors. There is a limit where additional processors increase the speed-up values. After a certain number of processors speed-up values does not increase. This phenomenon can be observed from the curve that belongs to 5% image density. As one can realize from Figure 4.4.(a)., speed-up does not increase when one moves from

16 processors to 31 processors. The reason for this is that the savings in computational time is offset by increased communication time for this image density. So for 5% image density, one may conclude that it is best to use 16 instead of 31 processors because going up to 31 processors does not change speed-up. For other image densities, going from 16 processors up to 31 processors increases the speed-up because they have more computational task than 5% image density. These image densities do not reach their flat portion of the curve in the processor range covered here.

Examining Figure 4.4.(a) and Figure 4.4.(b), one notices that the efficiencies for modified algorithm are uniformly more than those for standard algorithm in every image density. Thus one comes to the conclusion that the modified algorithms perform better than standard algorithms.

The improvement in speed-up values are given in Table 4.2.

Table 4.2. Percent improvement in speed-up due to Modified Algorithm.

No. of processors \ Image density	5%	10%	15%	20%	25%
4	2.4	1.1	0.7	0.5	0.3
7	4.9	2.3	1.5	1.0	0.7
11	6.6	3.0	1.8	0.9	0.5
16	14.3	7.7	5.1	3.3	2.5
31	-1.6	8.3	6.5	4.2	3.5

Examining Table 4.2., one can notice that percent improvement decreases as image density increases. In a given image density, percent improvement increases with increasing number of processors used.

Figure 4.5. gives the efficiency of the mesh modified algorithm. Note that efficiency increases with increasing image density for each case depicted. After a certain value of image density the efficiency no longer increases and remains flat. It is seen from Figure 4.5. that efficiency of the 31 node algorithm is the slowest to recover followed by 16 node algorithm. In general, Figure 4.5. shows that algorithms employing more processors need higher image densities to reach the stable efficiency point. Algorithms with high number of processors spend more on communication, thus the calculation task should be big enough to justify the communication time lost.

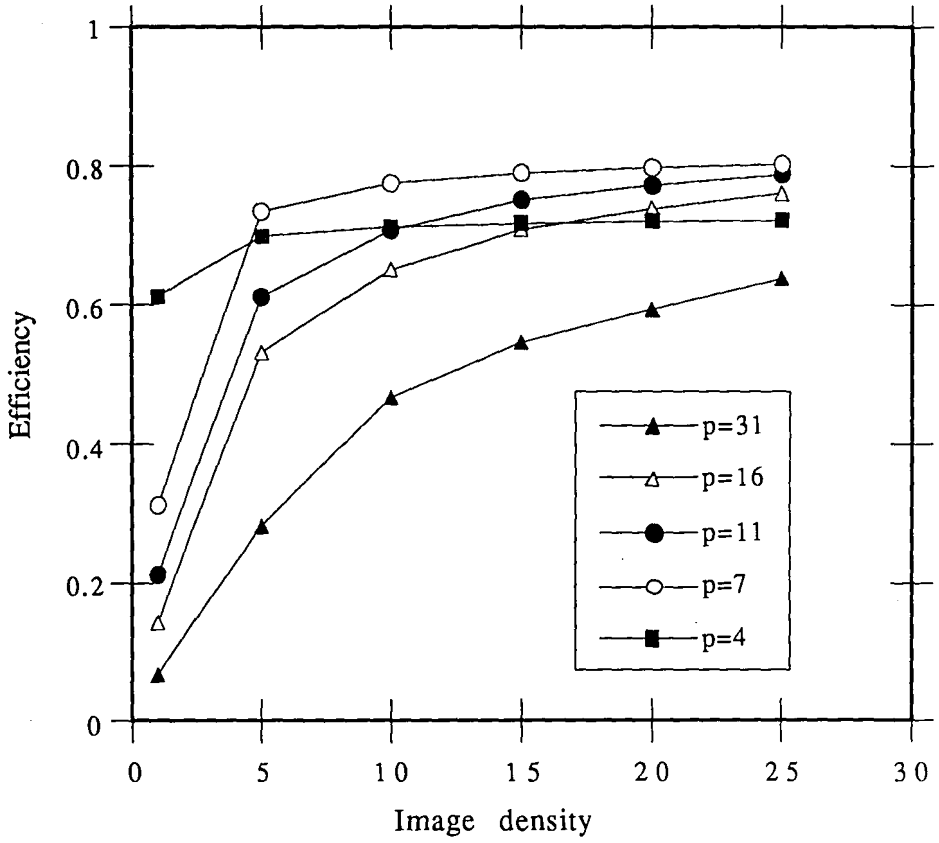

Figure 4.5. Dependence of efficiency on image density Mesh / Modified algorithm.

Chapter 5

Implementations on a Hypercube

5.1. Introduction

In this Chapter, the performance of the algorithms implemented on hypercubes of various sizes are presented. The results presented here are obtained by implementing algorithms on subcubes of NCUBE/10 hypercube.

A detailed description of hypercube topology was given in Chapter 2. Hypercube is extremely popular in parallel algorithms because of their versatility [21,22]. As mentioned earlier, it is possible to embed many structures in hypercubes. Linear array and mesh used in Chapter 3 and Chapter 4 were also embedded in subcubes of NCUBE/10 which are hypercubes. In this chapter, the algorithms are implemented on the hypercube itself.

Ranka and Sahni [4] have previously reported on a method to compute Hough Transform on hypercubes. Their method assumes that coordinates of pixels are already known, while the method presented here takes the binary image as input.

5.2. Implementation

Standard and modified algorithms were implemented in hypercubes with 4, 7, 11, 16 and 31 processors. The main idea behind the algorithms is similar to the linear array and mesh case. Processor 0 is assigned to the task of examining the image and sending the coordinate messages. The communication paths shown in Figure 5.1. make use of the hypercube structure. The dashed circle represents the image examining processor, the regular circle represents the parameter space generating processor and the coordinate messages travel in the direction of the arrows.

As has been mentioned earlier, hypercube topology has many superior characteristics. As a consequence, the communication paths shown in Figure 5.1. are the best of the three topologies considered in this thesis.

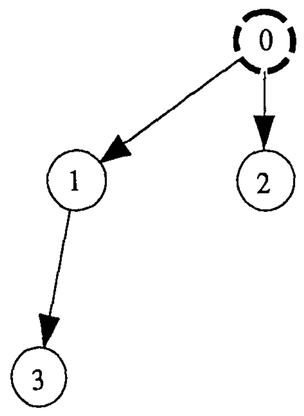

Figure 5.1.(a). Implementation of algorithm with 4 processors.

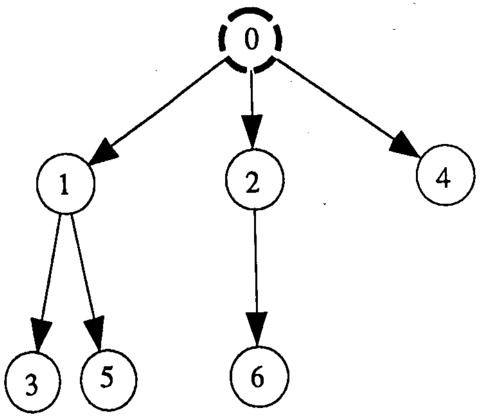

Figure 5.1.(b). Implementation of algorithm with 7 processors.

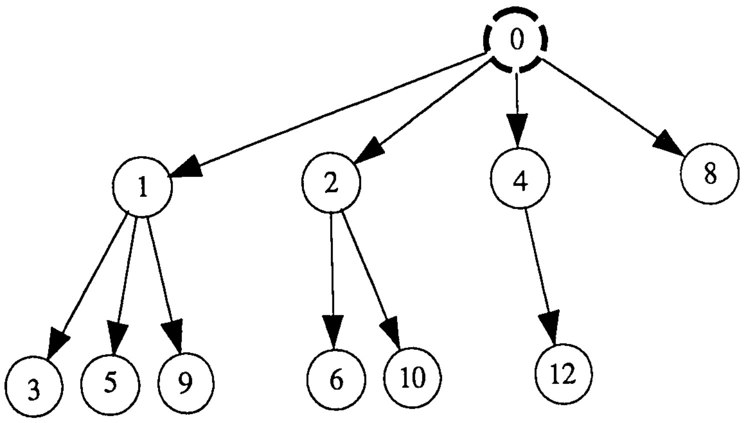

Figure 5.1.(c). Implementation of algorithm with 11 processors.

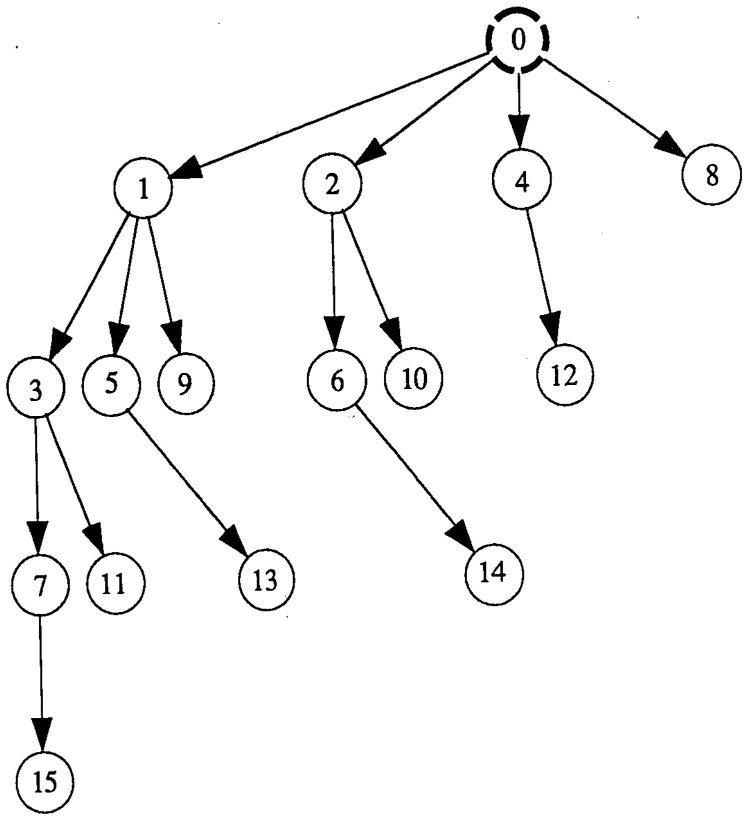

Figure 5.1.(d). Implementation of algorithm with 16 processors.

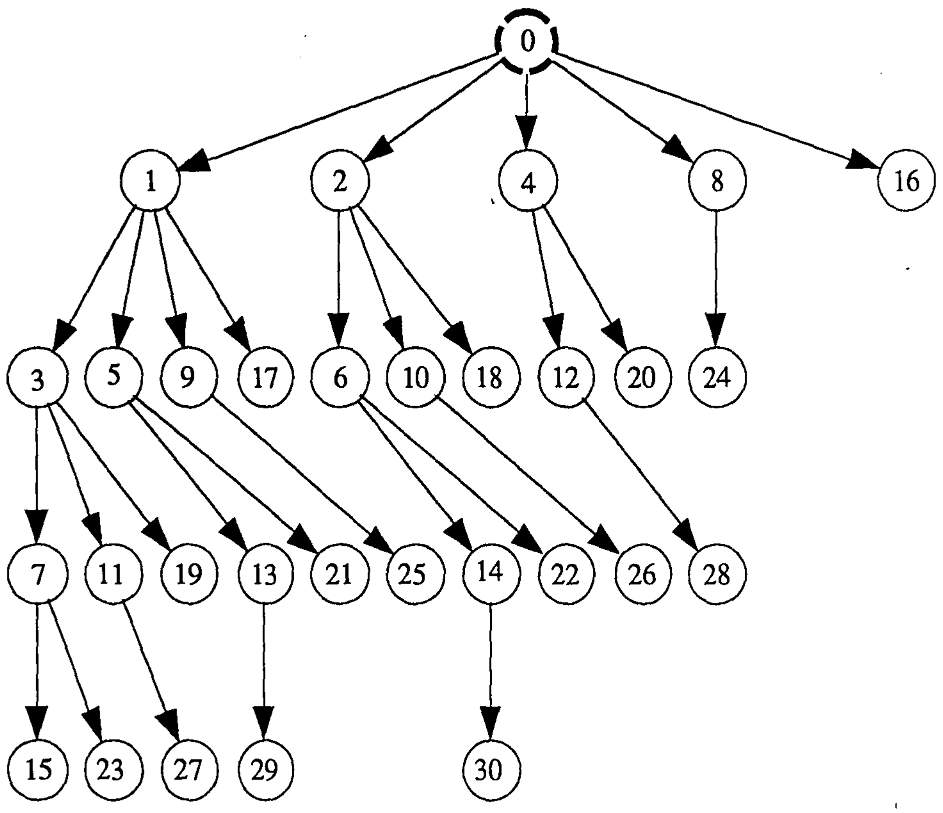

Figure 5.1.(e). Implementation of algorithm with 31 processors.

As one can notice from Figure 5.1.(e)., 31 node algorithm is implemented such that the maximum distance between processor 0 and any processor is 4. The comparable implementation of 31 node algorithm in linear array results in maximum distance of 30 as demonstrated in Chapter 2. Shorter communication path between processor 0 and the processor that is farthest from processor 0 means that the farthest processor receives the first coordinate message earlier which reduces the total Hough Transform time greatly because the farthest processor is completely idle before it receives the first coordinate message. One should also note that Hough Transform ends when the farthest processor finishes processing its last coordinate message.

5.3. Results

Figure 5.2.(a). gives the times obtained from Hypercube standard algorithm working with 31 processors. Figure 5.2.(b). is the corresponding graph for Hypercube modified algorithm.

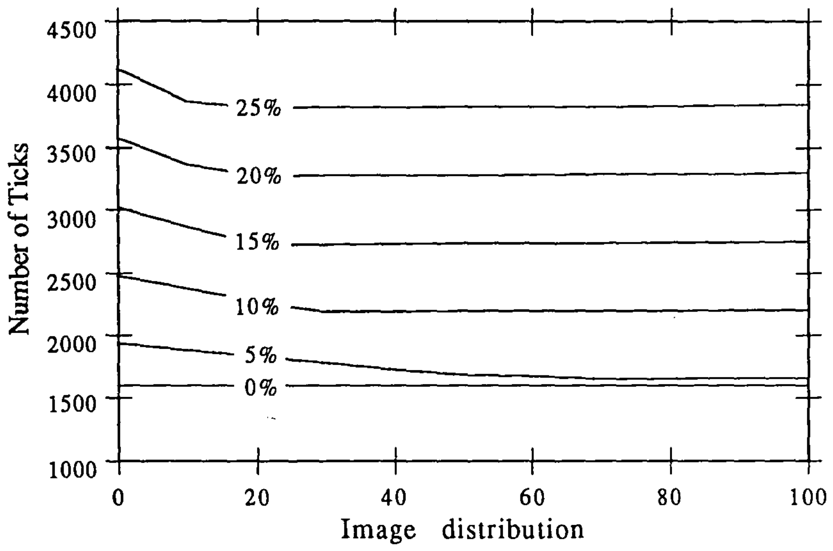

Figure 5.2.(a). Dependence of Time on image distribution Hypercube / Standard algorithm / 31 processors.

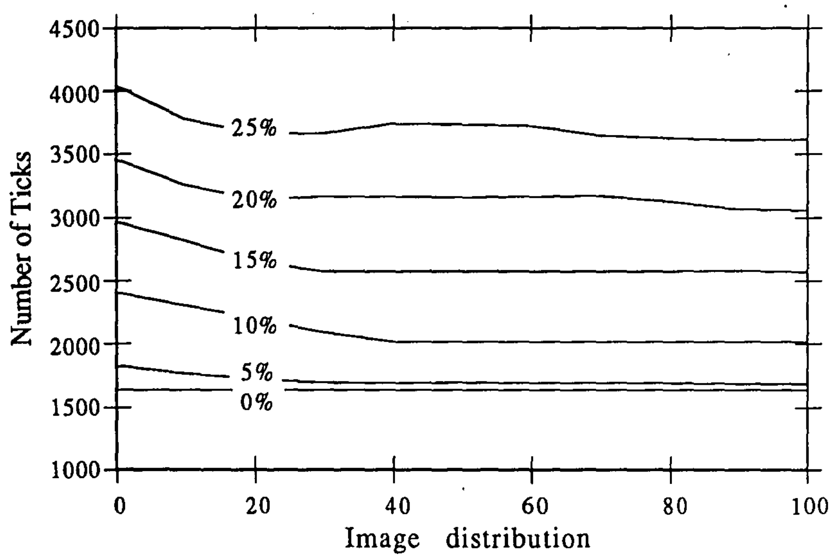

Figure 5.2.(b). Dependence of Time on image distribution Hypercube / Modified algorithm / 31 processors.

Figure 5.2. demonstrates that times obtained do not depend on image distribution as was the case with the other topologies. This very important characteristics of the algorithms presented in this thesis is also valid for the hypercube topology. Examining Figure 5.2. closely, one can notice that the times for image distribution cases where the most of the black pixels are in the second half of the binary image are a little higher. At this point, one should remember that 0% image distribution corresponds to the binary image whose first half is completely empty. In an algorithm that makes use of image distribution among processors this image results in half of the processors remaining "idle" so it nearly doubles the total time. In the algorithms presented here, it only leads to a increase of about 10%, which represents an enormous improvement.

Figure 5.3. illustrates the speed-up achieved by using the hypercube topology. The graphs look similar to the speed-up graphs in the previous Chapters as one expects. Here it should be noted that the algorithms implemented in hypercube topology produced the highest speed-up. This is due to the fact that the communication paths of the hypercube topology is the most efficient.

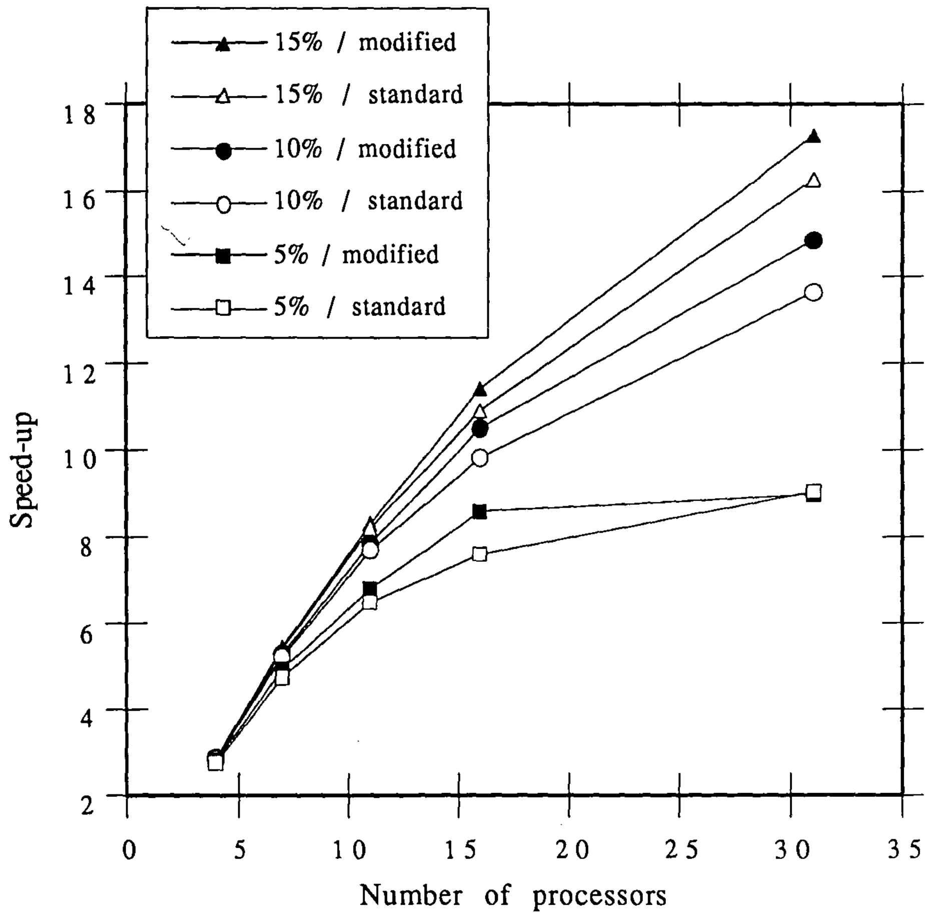

Figure 5.3.(a). Dependence of Speed-up on number of processors. Image densities: 5%, 10%, 15%.

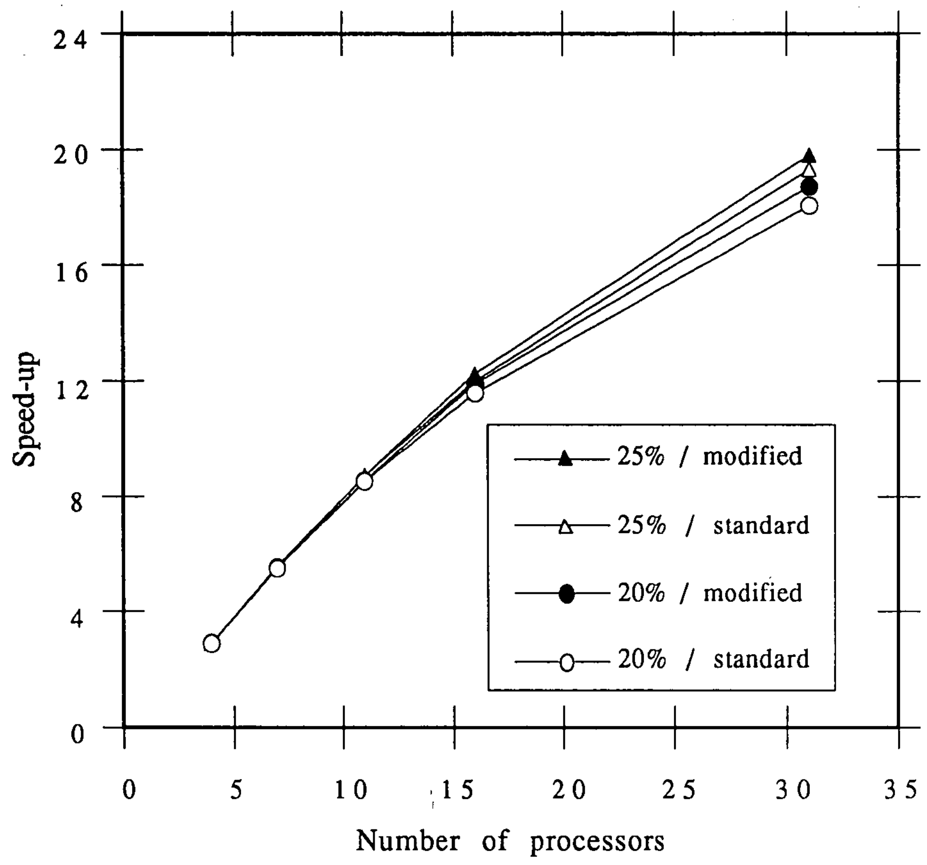

Figure 5.3.(b). Dependence of Speed-up on number of processors. Image densities: 20%, 25%.

Table 5.1. gives the percent improvement in speed-up achieved by using modified algorithm instead of using standard algorithm. The numbers in Table 5.1. resembles the numbers in the corresponding table for algorithms implemented in mesh. The numbers in Table 5.1. are generally lower than the numbers in the corresponding table for algorithms implemented in linear array. Algorithms implemented in linear array are not as efficient as the algorithms implemented in hypercube and consequently there is more room for improvement.

Table 5.1. Percent improvement in speed-up due to Modified Algorithm.

Image density / No. of processors	5%	10%	15%	20%	25%
4	2.6	1.0	0.7	0.4	0.3
7	3.8	1.6	1.0	0.5	0.4
11	4.8	2.0	1.1	0.3	0.0
16	13.0	6.9	4.6	2.9	2.1
31	-0.7	8.9	6.2	3.6	2.6

Table 5.1. illustrates that the difference between modified and standard algorithms are more emphasized in the case where processors are over-burdened with image. For example, 25% image density and 16 processor case represents a case where most of the time processors are busy with processing the previous message when the next coordinate message arrives. So in this case , there is no room for much improvement. However, if 16 processors and 5% image density case is considered, the improvement is noticed to be 13% much higher than 2.1% improvement for 25% image density.

Figure 5.4. illustrates the dependence of efficiency on image density in hypercube modified algorithm.

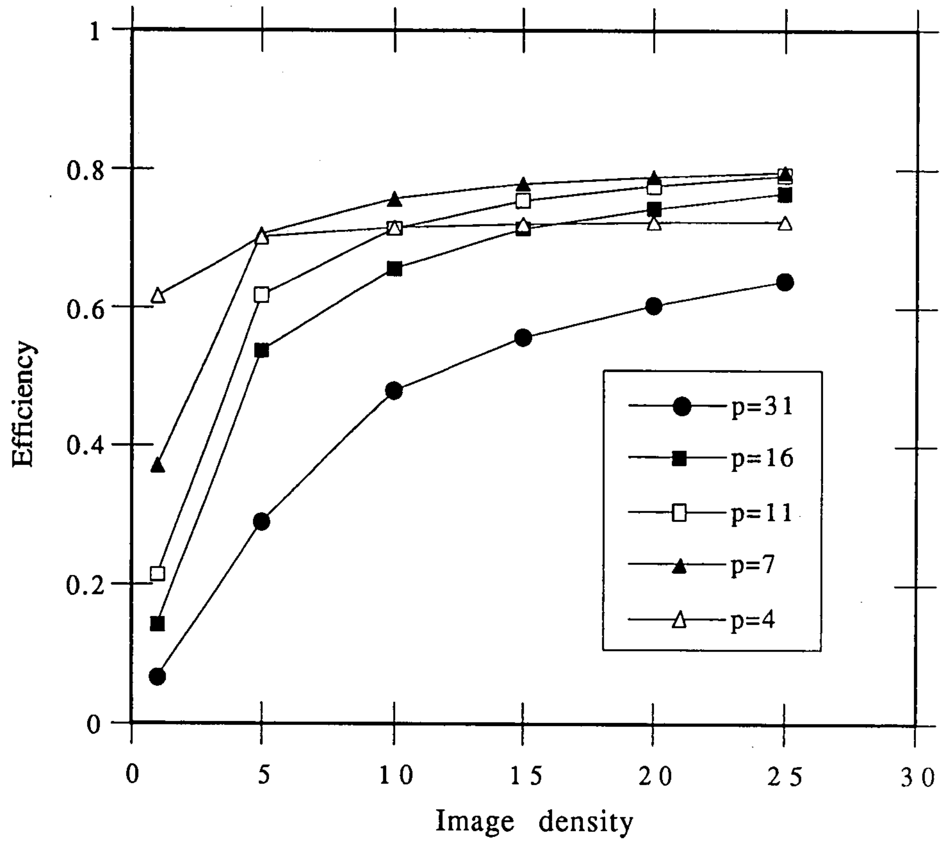

Figure 5.4. Dependence of efficiency on image density Hypercube / Modified algorithm.

Chapter 6

Conclusion

6.1. Discussion

In this thesis, two parallel algorithms for computing Hough Transform were developed and implemented on three different MIMD architectures, linear array, mesh and hypercube. Experiments were carried out by embedding these architectures in NCUBE/10 parallel machine at Lehigh University.

The results showed that the algorithms developed are not liable to asynchronism overheads, which result from nonhomogeneous distribution of edge pixels in binary images. Elimination of the asynchronism overheads was possible by the very nature of the algorithm structures without introducing additional computational effort. In literature, asynchronism overheads were reported to be responsible for 40% increase in total execution time, thus elimination of the asynchronism overheads proved to be a major achievement.

Our second "modified" algorithm using dynamic partitioning of the image proved to perform better than our "standard" algorithm in almost every case presented. The analysis of the algorithms on three different popular architectures provided a broader evaluation of the algorithm performances.

6.2. Future Directions

Separating the task of scanning the binary image from the task of generating the parameter space proved to provide a natural way to eliminate asynchronism overhead in Hough Transform. This idea behind the algorithms may be tried in other pattern recognition techniques.

Allocation of one processor to scan the image may be generalized to allocating m processors to scan the image, where m can be adjusted according to the requirements of the application.

Using dynamic partitioning of the binary image scanned also proved to useful in the modified algorithm provided in this thesis. The concept of dynamic partitioning may be generalized to dynamic task scheduling and it may be tried in a pattern recognition algorithm in the future.

References

[1] HUNGWEN Li, LAVIN Mark A., LE MASTER Ronald J. "Fast Hough Transform: A Hierarchical Approach" Computer Vision, Graphics, and Image Processing 36, 1986, pp 139-161.

[2] BEN-TZVI D., NAQVI A. A., SANDLER M. "Efficient Parallel Implementation of the Hough Transform on a distributed memory system" Image and Vision Computing, Vol 7, No 3, August 1989.

[3] KANNAN, C. S. CHUANG, Henry Y. H. "Fast Hough Transform on a Mesh Connected Processor Array" Information Processing Letters 33, January/10/1990, pp 243-248.

[4] RANKA, Sanjay SAHNI, Sartaj "Computing Hough Transforms on Hypercube Multicomputers" The Journal of Supercomputing 4, 1990, pp. 169-190.

[5] THAZHUTHAVEETIL, Matthew J. SHAH, Anish V. "Parallel Hough Transform Algorithm Performance" Image and Vision Computing Vol 9 No 2 April 1991

[6] HOUGH, P. V. C. "Methods and means for recognizing complex patterns" US patent no. 3069654 (1962).

[7] BALLARD, D. H. "Generalizing The Hough Transform to Detect Arbitrary Shapes" Pattern Recognition Vol 13 No 2 pp. 111-122 1981

[8] QUAN, Long MOHR, Roger "Determining perspective structures using hierarchical Hough Transform"
Pattern Recognition Letters, May 1989, pp. 279-286.

[9] DAVIES, E. R. "Occlusion Analysis for Object Detection using the Generalized Hough Transform" Signal Processing 16, 1989, pp. 267-277.

[10] TSUI, H. T. CHAN C. K. "Hough Technique for 3D Object Recognition" IEE Proceedings, Vol. 136, Pt. E , No 6, November 1989.

[11] DAVIES, E. R. "A Modified Hough Scheme for general circle location" Pattern Recognition Letters 7, 1988, pp. 37-43.

[12] SEITZ, C. L. "The Cosmic Cube," Comm. ACM, Vol. 28, No.1, January, 1985, pp. 22-33.

[13] KUMAR, Prasanna V. K. TSAI, Yu-Chen "Designing Linear Systolic Arrays" Journal of Parallel and Distributed Computing Volume 7, pp. 441-463 1989.

[14] O'HALLARON, David R. "Uniform Approach for Solving Some Classical Problems on a Linear Array" IEEE Transactions on Parallel and Distributed Systems Volume 2, No. 2, April 1991.

[15] LEE, P. KEDEM, Z. "Synthesizing Linear Array Algorithms from Nested For Loop Algorithms" IEEE Transactions on Computers Volume 37, pp. 1578-1598, December 1988.

[16] BAHETI, R. S. O'HALLARON, D. R. ITZKOWITZ, H. R. "Mapping Extended Kalman Filters onto Linear Arrays" IEEE Trans. Automat. Contr., 1990.

[17] DEHNE, Frank HASSENKLOVER, Anne-Lise SACK, Jorg-Rudiger "Computing the configuration space for a robot on a mesh-of-processors" Parallel Computing Vol 12 pp. 221-231 1989

[18] HOLEY, Andrew J. OSCAR, H. Ibarra "Iterative algorithms for the planar convex hull problem on mesh-connected arrays." Parallel Computing Vol 18 pp. 281-296 1992

[19] SCHERSON, Isaac D. CORBETT, Peter F. "Communications Overhead and the Expected Speedup of Multidimensional Mesh-Connected Parallel Processors" Journal of Parallel and Distributed Computing Vol. 11 pp. 86-96 1991

[20] KANNAN, C. S. CHUANG, Henry Y. H. "Fast Hough Transform on a Mesh Connected Processor Array" Information Processing Letters Vol. 33 pp.243-248 1990.

[21] RANKA, Sanjay SAHNI, Sartaj "Image Template Matching on MIMD Hypercube Multicomputers" Journal of Parallel and Distributed Computing Vol 10 pp.79-84 1990.

[22] CHEN, Ming-Syan SHIN, Kang G. "Subcube Allocation and Task Migration in Hypercube Multiprocessors" IEEE Transactions on Computers Vol 39 No 9 September/1990.

Vita

The author was born on January/01/1967 in Mugla, Turkey. He is the son of Mehmet Ali OZBEK and Hosgun OZBEK.

In June-1985 , he received his high school diploma from Ankara Fen Lisesi, Ankara, Turkey. On July/03/1992, he received Bachelor of Science degree in Electrical Engineering from Bogazici University, Istanbul, Turkey. In Fall of 1989, he started his studies in Lehigh University pursuing Master of Science degree in Electrical Engineering.

The author is a member of IEEE and ACM.